INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

UNIVERSITY OF CALIFORNIA, SAN DIEGO

Floorplanning Algorithms for VLSI Physical Design Automation

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in Computer Science

by

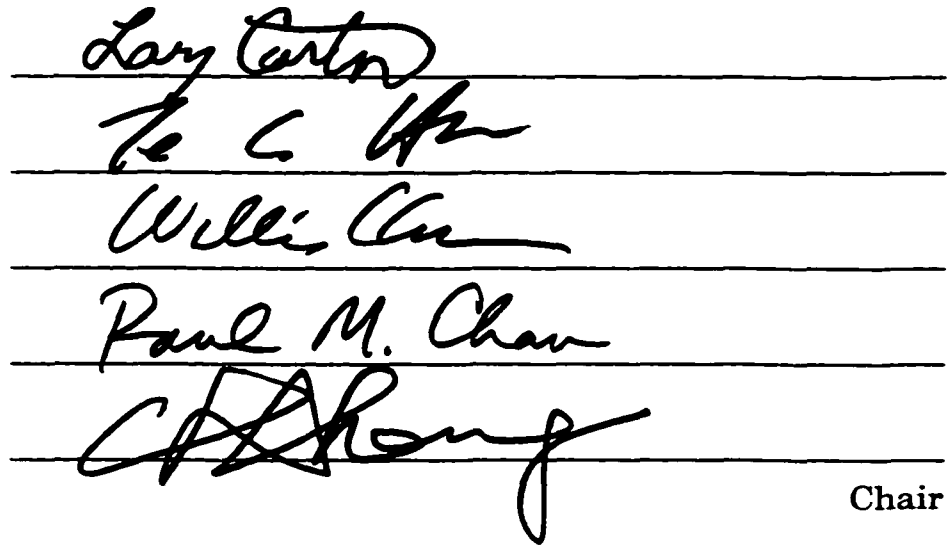
Yingxin Pang

Committee in charge:

Professor Chung-Kuan Cheng, Chair
Professor Lawrence Carter
Professor Te. C. Hu
Professor Paul Chau
Professor Walter H. Ku

2000

The dissertation of Yingxin Pang is approved, and it is acceptable in quality and form for publication on microfilm:



Chair

University of California, San Diego

2000

To my parents

and

Yukuang, Mingming and Tangtang

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank my advisor, Chung-Kuan Cheng, for all his support, encouragement and guidance over the past years. Professor Cheng's insight and advice have made an enormous contribution to this dissertation. I have enjoyed having the opportunity to work with him.

Additionally, I thank the members of my committee, Professors Larry Carter, T. C. Hu, Paul Chau, and Walter H. Ku, for helpful suggestions and support. Professors Scott Baden, James Bunch, and Josef Rinott also gave invaluable assistance in my study in both Mathematics department and CSE department.

Thanks to the other members of the lab, whose friendship and support helped make my graduate career fun. In particular, Xiaodong Yang gave me a lot of help on the Framemaker formatting of the dissertation.

Special thanks to my parents, who help me taking care of my sons during my school years, and certainly deserve most of the credit for any accomplishments of mine.

Finally, I dedicate this dissertation to my husband Yukuang, my sons Mingming and Tangtang. Yukuang's unwavering support and encouragement were my greatest resources during these past years. Mingming's and Tangtang's smile makes me joyful every day.

Portion of Chapter II, is a reprint of the material as it appears in "An Enhanced Perturbing Algorithm for Floorplan Design Using the O-tree Representation" Yingxin Pang; Chung-Kuan Cheng; Takeshi Yoshimura, Proceedings of international Symposium on Physical Design, 2000.

Portion of Chapter III, is a reprint of the material as it appears in "Block Placement with Symmetry Constraints Based on the O-tree Non-Slicing Representation", Yingxin Pang; Florin Balasa; Koen Lampaert; Chung-Kuan Cheng, 37th. Proceedings of ACM/IEEE Design Automation Conference, 2000.

This dissertation was funded in part by grants from National Science Foundation and California Micro Project.

VITA

August 19, 1966	Born, Dalian, P. R. China
1988	B.S. Applied Mathematics Shanghai Jiao Tong University
1991	M.S. Applied Mathematics Dalian University of Technology
1996	M.S. Statistics University of California, San Diego
2000	Ph.D. Computer Science University of California, San Diego

PUBLICATIONS

Y. Pang, F.J. Liu, C.K. Cheng, "A new Method for Hierarchical Circuit Composition", ACM, *Intn'l Workshop on Timing issues in the Specification and Synthesis of Digital Systems*, pp. 71-76, 1999.

Y. Pang, C.K. Cheng T. Yoshimura, "An Enhanced Perturbing Algorithm for Floorplan Design Using the O-tree Representation", *ACM Proc. ISPD*, pp. 168-173, 2000.

Y. Pang, F. Balasa, K. Lampert, C. K. Cheng, "Block Placement with Symmetry Constraints in the Context of O-tree Representation", *Proc. 37th. ACM/IEEE Design Automation Conf.* , June, 2000

FIELDS OF STUDY

Major Field: Computer Science

Studies in VLSI Computer-Aided-Design
Professor Chung-Kuan Cheng

ABSTRACT OF THE DISSERTATION

Floorplanning Algorithms for VLSI Physical Design Automation

by

Yingxin Pang
Doctor of Philosophy in Computer Science
University of California, San Diego, 2000
Professor Chung-Kuan Cheng, Chair

The increase in complexity of very large scale integrated (VLSI) systems has imposed many new challenges for physical design automation. Floorplanning is a critical phase in the physical design cycles since the overall quality of the layout, in terms of area and performance is mainly determined in this phase.

Floorplanning is to pack all the circuit elements in a chip without violating design rules, so that the circuit performs well and the production yield is high. Many existing layout design algorithms, however, are not able to scale well to match the rapid increase in design complexity. The objective of this dissertation is to develop efficient floorplanning algorithms for VLSI designs.

The O-tree representation has recently gained increasing interest: compared to other representations (slicing structure, sequence-pair, BSG), O-trees need a smaller amount of encoding storage and linear time computation effort to generate each placement configuration. In addition, the upper-bound of possible encodings is smaller, therefore, a smaller search space for any optimization algorithm. These important advantages are strong incentives for us to address the floorplanning problems in the context of the O-tree representation.

We present an efficient algorithm to handle the core floorplanning problem. Our main algorithm--the enhanced perturbing algorithm performs a

greedy perturbation each time and try to reach a local optimum very quickly. It can be used to improve an existing floorplan and can be extended to carry out a complete floorplan design.

We address the problem of handling symmetry constraints. This problem arises in analog placement, where symmetry is often used to match layout-induced parasitics and to balance thermal couplings in differential circuits. We add these additional constraints in the representation of general blocks. When we explore the space of the O-trees, we only consider the symmetric-feasible O-trees since only symmetric-feasible O-trees can lead to consistent placements satisfying the O-tree constraints and the symmetric constraints.

We extend the O-tree approach to handle rectilinear blocks. First we explore the properties of L-shaped blocks, then decompose arbitrarily shaped rectilinear blocks into a set of sub-L-shaped-blocks. The properties of L-shaped blocks can be applied to general rectilinear blocks.

We applied our algorithms to several test cases. The good performance of our placement tools proves the effectiveness of our technique.

Chapter I: Introduction

The increase in complexity of very large scale integrated (VLSI) systems has imposed many new challenges for physical design problems. Floorplanning and Placement is a critical phase in the physical design cycles since the overall quality of the layout, in terms of area and performance, is mainly determined in this phase. The input to the floorplanning and placement phase is a set of blocks. The blocks for which the dimensions are known are called hard blocks, and the blocks for which dimensions are yet to be determined are called soft blocks. The problem of assigning locations to all blocks on a layout surface is called the Floorplanning and Placement Problem. In this dissertation, we use the terms *floorplanning* or *placement* as the general terminology for Floorplanning and Placement Problem.

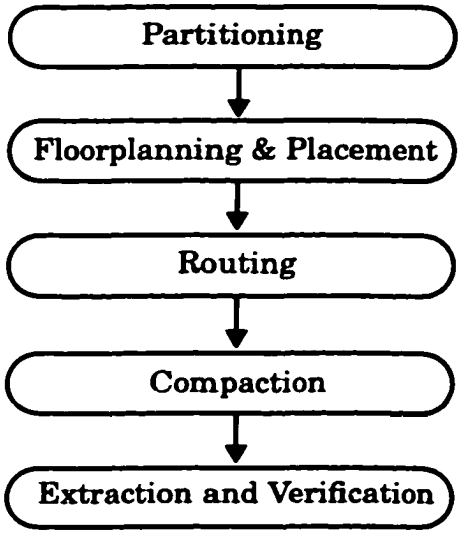


Figure I.1: Physical design cycle

I.1 Problem Formulation

Floorplanning in VLSI physical design requires packing all the circuit elements in a chip without violating design rules, so that the circuit performs well and

the production yield is high. There is a great variety of targets in different problems but the following problem which we refer to as the core floorplanning problem.

Given a set S of n rectangular modules $S = \{1,2,...,n\}$, associate with each block a list of width and height tuples (w, h) and a netlist between the blocks. The output is an assignment of coordinates to the lower left corners of the rectangular blocks such that there are no two rectangular blocks overlapping and the objective function is minimized. The objective function could be the total area of the floorplan, the total wire length of the floorplan, or the weighted sum of the two quantities.

Rectangle packing is an NP-hard problem. A typical approach is to introduce a solution space and search the optimal implementation corresponding to the objective function in the solution space. The solution space must have the following four properties:

(1) The solution space is finite

(2) Every solution can lead to a consistent floorplan construction

(3) Transforming a solution to a floorplan is possible in polynomial time

(4) There exists a solution corresponding to one of the optimal floorplans

However since the size of any such solution space is expected to be exponential, an exhaustive search of the whole space will take too much time. Several heuristics have been proposed to find a good solution in a moderate time, such as, simulated annealing and genetic algorithms. Given a time limit, such a heuristic stops the search half-way and outputs the best solution found so far.

I.2 Topological Representation

Using the topological model to represent a floorplan solution is an important and fundamental method. The cell positions are specified relatively, based on the topological relations between them and the solution space includes all possible topologies. In general, there are two kinds of topological structures: slicing and non-slicing structures. If a floorplan can be obtained by recursively cutting a rectangle into two parts by either a vertical or a horizontal line, the floorplan has a slicing structure. Otherwise, the floorplan has a non-slicing structure.

Slicing structure provides a simple way for optimizing the block orientations, defining reasonable channels in global routing, and appropriately ordering the channels during detail routing. But the slicing structure is very limited since most of the dissections are nonslicing. This can degrade layout density, especially when cells are very different in size, which is often the case in practice. On the other hand, with the increase of routing layers, channel routing is being replaced by area routing; blocks are packed together to minimize the area. As such, the wasted area due to slicing structures is more evident. Therefore the non-slicing structure is more favorable in today's floorplanning design.

I.2.1 Slicing Structure

A floorplan is a slicing if either it is a basic rectangle or there is a line segment that divides the enclosing rectangle into two pieces such that each piece is a slicing. Another useful way to describe a slicing floorplan is to specify the natural hierarchical structure in an oriented rooted binary tree called a slicing tree. Each internal node of the tree is labeled either *"V"* or *"H"*, corresponding to either a vertical or a horizontal cut, respectively. Each leaf corresponds to a basic rectangle and is labeled by a number between *1* and n when the slicing structure has n basic rectangles.

When we traverse a slicing tree in post-order, we obtain a sequence formed by the nodes of the tree. This sequence is a Polish expression[26]. Figure I.2 gives a slicing dissection, and its corresponding slicing tree. Its Polish expression representation is *abVcVefHdVH*. Since there is only one way of performing a post-order traversal of a binary tree, there is a one to one correspondence between slicing trees and their corresponding Polish expressions.

The slicing structure is widely used in industry since focusing only on slicing floorplans significantly reduces the search space and leads to fast running time. Secondly, the shape flexibility of the modules can be fully exploited to pack modules tightly using an efficient shape curve computation technique.

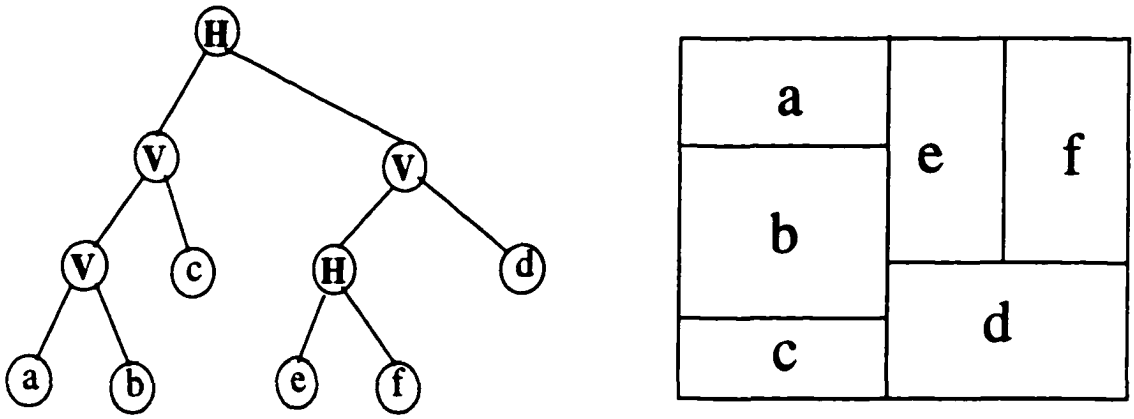


Figure I.2: Slicing structure

I.2.2 Sequence Pair

Recently, several nonslicing topologies have been proposed. Murata *et al* [16] suggested encoding the "left-right" and "up-down" positioning relations between cells using two sequences of cell permutations, named sequence-pairs. Nakatake *et al* [18] devised a meta-grid structure (called bounded-sliceline grid or BSG) to define orthogonal relations between modules without using physical dimensions. The sequence-pair is a popular nonslicing topological representation since it is adequate to handle high-performance analog layout.

A sequence-pair for a set of n modules is a pair of permutations of the n module names. For example, *(abdecf, cbfade)* is a sequence-pair for module set *{a,b,c,d,e,f}*. The variety of the sequence-pair for n modules is $(n!)^2$.

Given a sequence pair for n blocks, an n oblique grid can be constructed as shown in Figure I.3:

45° slope lines are named from left to right by the symbols in the first sequence, and -45° slope lines are similarly named by the symbols in the second sequence. Each block is placed at the cross of the two slop lines which are named by the same symbol corresponding to the block. The plane can be divided by the two crossing slope lines into four quarters for any block.

It imposes horizontal and vertical constraints for every pair of modules as follows:

If module *b* is in the right quarter view range of module *a* on the oblique grid, then *b* should be placed to the right of *a*. If module *b* is in the top quarter view range of module *a* on the oblique grid, then *b* should be placed above *a*.

The horizontal and vertical constraint graphs can be constructed as shown in Figure I.3. Each vertex corresponds to a block, there is an arc from block *a* to block *b* if and only if *b* is right to *a*. Each vertex has a weight which equals the width of the corresponding block. The vertical constraint graph can be similarly derived.

Given the horizontal and vertical constraints graphs, one of the optimal packings under the constraints can be obtained in $O(n^2)$ time by applying the well-known longest path algorithm for vertex weighted directed acyclic graphs. Since the width and the height of the chip is independently minimized, the resultant packing is the best of all the packings under the constraints.

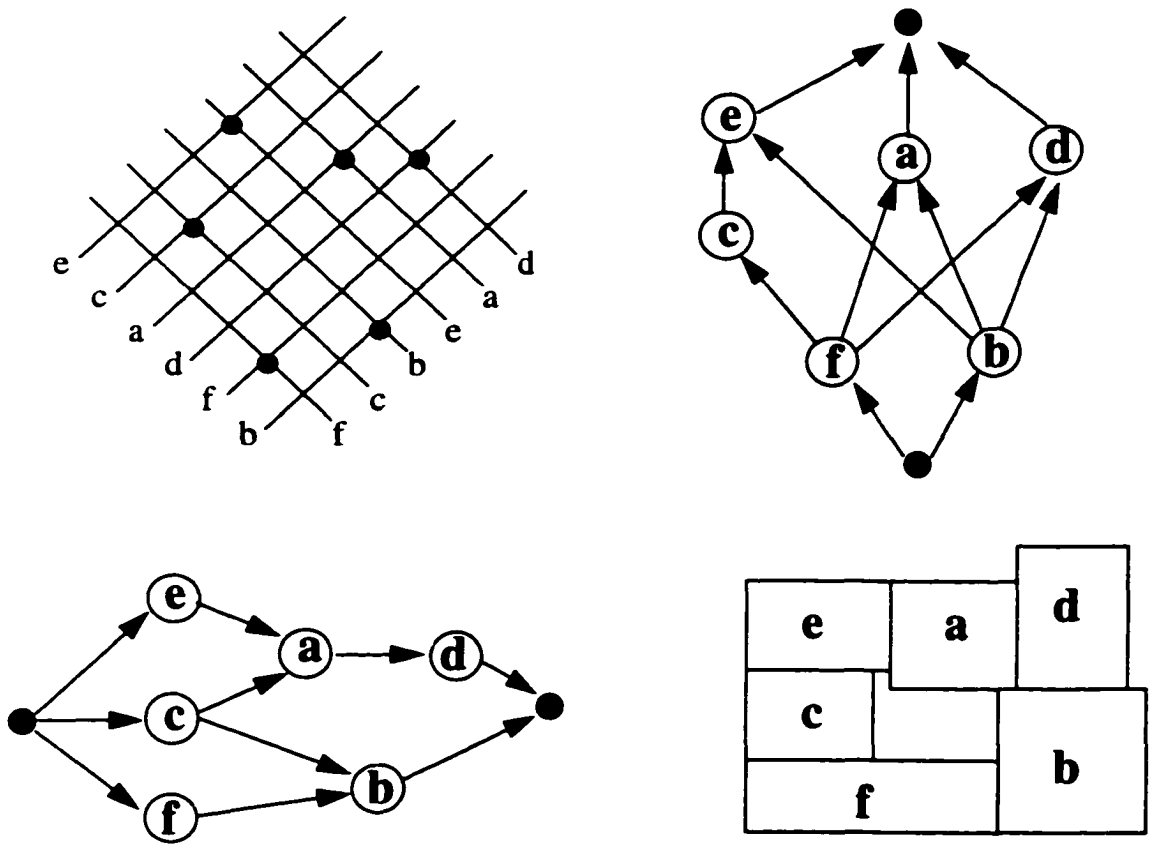


Figure I.3: Sequence-pair

For every packing, there is a sequence-pair corresponding to it. For every sequence-pair, the corresponding packing can be obtained in $O(n^2)$.

I.2.3 O-tree

More recently, the ordered tree--O-tree representation has been proposed by Guo *et al.* [7] An n-node O-tree is a tree with $n+1$ nodes encoded by (T, π), where T is a $2n$-bit string that identifies the branching structure of the tree, and π is a permutation of the n node labels (excluding the root). When traversing the tree, we write a '0' for descending an edge and a '1' for subsequently ascending that edge. Given the 7-node O-tree in Figure I.4, we can represent it as *T= 00110100011011*, π=*abcdefg*.

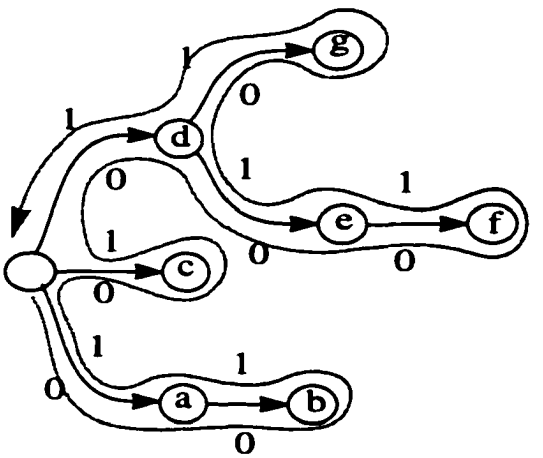


Figure I.4: O-tree

Compared with sequence pair and slicing structures, the O-tree representation has the following advantages:

O-tree takes only $n(2 + \lg n)$ bits to describe, where n is the number of blocks. Note that a sequence pair takes $2n \lg n$ bits. Even a binary tree for slicing structure takes $n(6 + \lg n)$ bits.

Transforming an O-tree to its representing placement is linear to the number of blocks, i.e. $O(n)$. For a sequence pair, its takes $O(n \lg n)$ operations to construct the placement. Note that it also takes $O(n)$ operations to construct a slicing structure from a binary tree.

Given a placement, there is an O-tree representation. The number of combinations of an O-tree is $O(n!\, 2^{2n-2}/n^{1.5})$. This is very concise compared with the sequence pair. The number of combinations of a sequence pair is $O((n!)^2)$. The ratio of the complexity between sequence pair and O-tree is approximately $O(n^2\ (n\ /\ 4e)^n)$. The complexity of O-tree is even lower than a binary tree structure for slicing floorplan which has $O(n!\, 2^{5n-3}/n^{1.5})$ combinations.

An O-tree where nodes represent rectangular blocks imposes both horizontal and vertical positioning constraints:

- all children blocks must be placed to the right of their parent;
- if two blocks are overlapping along their x-coordinate projection, the block having a higher index in permutation π must be placed above the one having a lower index.

It has been proven that a minimum area packing under the O-tree positioning constraints can be obtained in linear time and, moreover, there is an O-tree from which a global optimal packing can be derived.

If B_1, B_2, ... , and B_n are the blocks to be placed on a chip, assume each B_i is a rectangle, having an associated width w_i and height h_i, and having (x_i, y_i) as the coordinates of its left-bottom corner. To construct a minimum area placement for an O-tree, the horizontal positioning constraints implies that:

if B_i is the parent of B_j, then $x_j = x_i + w_i$. The root may be viewed as the left chip boundary, $x_{root} = 0$ and $w_{root} = 0$.

According to the vertical positioning constraints:

for each block B_i, let $\psi(i)$ be the set of blocks B_k which have been placed on the chip and whose spanning interval $(x_k, x_k + w_k)$ overlaps $(x_i, x_i + w_i)$. Then

$$y_i = max_{\,k \in \psi(i)}\, y_k + h_k \qquad \text{if } \psi(i) \text{ is non-empty}$$
$$= 0 \qquad \text{otherwise}$$

Figure I.5 shows a corresponding placement of the O-tree in Figure I.4.

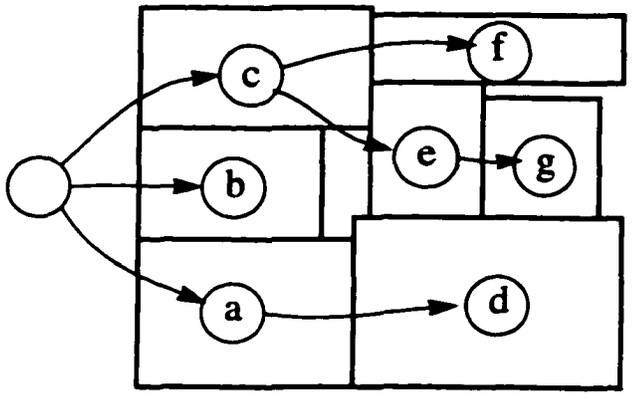


Figure I.5: An O-tree placement

I.3 Motivation

Rapid decrease in device dimensions leads to much higher design complexity. Many existing layout design algorithms, however, are not able to scale well to match the rapid increase in design complexity. The objective of this dissertation is to develop efficient and highly scalable floorplanning algorithms for VLSI designs.

The O-tree representation has recently gained increasing interest: compared to the other topological representations mentioned above (sequence-pair, BSG), O-trees need a smaller amount of encoding storage and linear time computation effort to generate each placement configuration. In addition, the upper-bound of possible encodings is smaller, which entails a reduced representation redundancy and, therefore, a smaller search space for any optimization algorithm solving a minimum area packing problem. These important advantages are strong incentives to address the floorplanning problems in the context of the O-tree representation.

In addition to the general core floorplan problem, there are several important factors that have to be added to the basic problem in chip floorplanning. The factors are

(1) Flexibility of the blocks: The blocks are rectangles, but there are multiple width and height tuples associated with each block.

(2) Symmetry considerations: In high performance analog circuits, it is often required that groups of devices be placed symmetrically with respect to one or more symmetry axes. Symmetric placement allows for symmetric routing and results in matched parasitics.

(3) Shape of the blocks: Recent advances in sub-micron technology makes it possible to realize a big system on a single chip. Designing such a huge VLSI layout is hard and so design reuse has been attracting much interest. In the layout design of such VLSI systems, blocks are not often simple rectangles. Therefore packing algorithms for arbitrary shaped rectilinear blocks are desirable.

All versions of the floorplanning problems are NP-hard. Thus the additional additions make the hard problem even harder. In this dissertation, we extends the O-tree structure to handle these problems. In theory, our techniques show less complexity than other methods. In practice, the experimental results demonstrate the efficiency of our techniques.

I.4 Organization of the Dissertation

The dissertation presents algorithms to handle the floorplanning problems involved in designing very large scaled integrated circuits.

In Chapter II, we present an efficient algorithm to handle the core floorplanning problem. Our main algorithm--the enhanced perturbing algorithm performs a greedy perturbation each time and try to reach a local optimum very quickly. It can be used to improve an existing floorplan and can be extended to carry out a complete floorplan design. In order to obtain a good placemnt, we thoroughly test all possible rotation and flipping configurations of each block and insert it in the best configuration each time. To demonstrate the efficiency of the algorithm, we apply our algorithm to MCNC benchmark circuits. Experimental results show that the algorithm obtains excellent results in much less time than other techniques.

Chapter III addresses the problem of handling symmetry constraints in the context of the O-tree representation. This problem arises in analog placement, where symmetry is often used to match layout-induced parasitics and to balance thermal

couplings in differential circuits. We add these additional constraints in the representation of general blocks. A necessary and sufficient condition of a symmetric-feasible O-tree is given by the properties of the horizontal and vertical constraint graphs. When we explore the space of the O-trees, we only consider the symmetric-feasible O-trees since only symmetric-feasible O-trees can lead to consistent placements satisfying the O-tree constraints and the symmetric constraints. The good performance of our placement tool in dealing with several analog designs taken from industry proves the effectiveness of our technique.

Chapter IV extends the O-tree approach to handle rectilinear blocks. First we explore the properties of L-shaped blocks, then decompose arbitrarily shaped rectilinear blocks into a set of sub-L-shaped-blocks. The properties of L-shaped blocks can be applied to general rectilinear blocks. The experiment results show that the algorithm achieves placements with excellent area utilization.

Chapter V concludes the dissertation and suggests avenues for future investigation.

Chapter II: An Enhanced Perturbing Algorithm for Floorplan Design

II.1 Introduction

There are many problems in the natural world which resemble NP-hard optimization problems. The simulation based algorithms simulate some of such natural processes or phenomena. Simulated annealing (SA) was the first to be proposed and is still the most widely applied. It simulates the annealing process which is used to temper metals. It is well known that, at high temperatures, a system can be found equally likely in any one of its available states, while at sufficiently low temperatures, only the minimum energy states can be reached,

The generality of SA has made it an important tool to approach NP-hard optimization problems. However, the initial and final annealing temperatures, as well as the cooling rate, are important implementation parameters, and it demands an enormous computational effort--a fact that has spurred the development of faster and more efficient alternatives.

An algorithm called the deterministic algorithm [7], based on the O-tree representation, is used to iteratively improve an existing floorplan and has obtained good results. For an initial O-tree with n blocks, the deterministic algorithm iteratively perturbs each block and, for each block, it tries all possible insertion positions in the tree. For each insertion position, it constructs a corresponding placement and accept the best placement. Since the placement construction of an O-tree takes $O(n)$ operations and the number of possible insertion positions is $O(n)$, it takes $O(n^2)$ for one block. Overall the deterministic algorithm runs in $O(n^3)$.

In this chapter, we describe an algorithm which achieves the same goal of the deterministic algorithm with an increase in speed of one order of magnitude. Like the deterministic algorithm, it iteratively perturbs each block. For each block, however, it follows a depth first search sequence and reduces the checking of each position to constant time by amortizing the checking of other positions in that sequence. Our algo-

rithm finds the best insertion position without constructing all the placements. Therefore our algorithm takes only $O(n)$ for each block. The whole algorithm runs in $O(n^2)$. Our algorithm can be used to improve an existing floorplan, or to carry out a complete design of a floorplan. To demonstrate the efficiency of our algorithm, we apply it to five MCNC benchmarks and compare it with the deterministic algorithm. The results show that we produce the same high quality results with significantly less CPU time.

The rest of this chapter is structured as follows: in section II.2, we give a brief review of the deterministic algorithm. In section II.3, we present our enhanced perturbing algorithm with the objective function restricted to be the total area. In section II.4, we extend the objective function to consider the wire length. In section II.5, we illustrate how to carry out the complete floorplan design by our algorithm. In section II.6, the experimental results are shown. A summary is given in section II.7.

II.2 The Deterministic Algorithm

Given an O-tree, a perturbation to that configuration can be made by deleting a block from the O-tree and inserting it in other positions of the O-tree. In simplicity, we constrain our insertion positions to be the external nodes of the tree. If an initial O-tree has n blocks, then after deleting the perturbed block, the number of the possible insertion positions as external nodes is $2n-1$ (it may include the previous position of the block). Figure II.2 shows the resulting O-tree obtained by deleting block a from the original O-tree shown in Figure II.1 and all the possible insertion positions. We note that the number of blocks in the original O-tree is 6, while the number of possible inserting positions as external node is 11.

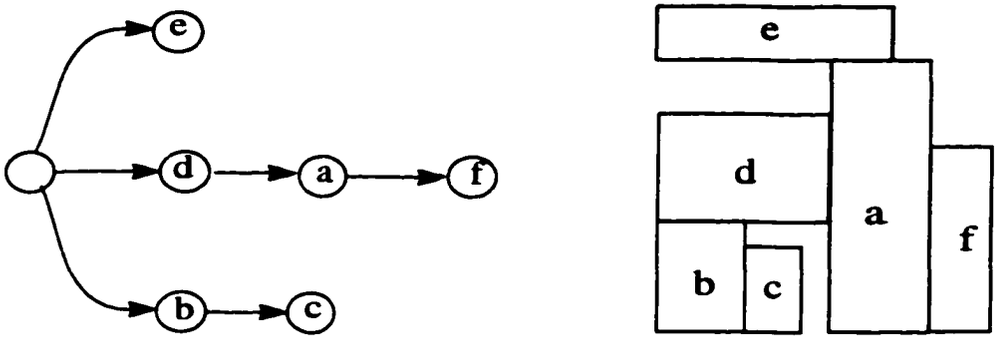


Figure II.1: An O-tree and its placement

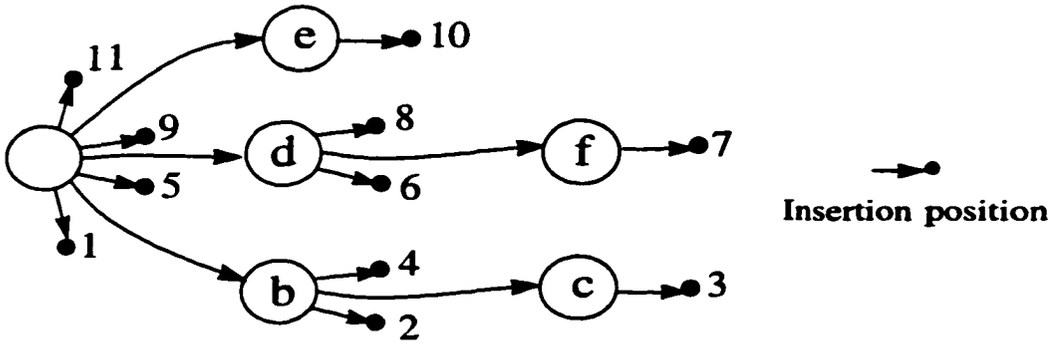


Figure II.2: Possible insertion positions

The different insertion positions can create different O-tree configurations. The deterministic algorithm attempts to reconstruct the O-tree by placing the deleted block in the position which the corresponding placement decreases the cost function the most. Therefore the original O-tree is iteratively improved by perturbation.

The basic procedure of the **Deterministic Algorithm** is as follows:

given an initial O-Tree OT

for each block b in OT

set min_cost = infinite

remove b from OT, so that a resulting O-tree OT_1 is obtained

for each possible position p in OT_1 for b

```
        set OT2 = new O-Tree inserting block b in position p
        build the placement of OT2
        set c = cost (OT2)
        if c < min_cost then
          set min_cost = c
          set min_OT = OT2
        end if
      end for
      set OT = min(cost(OT), cost(min_OT))
    end for
    Output placement for OT
```

There are two "**for**" loops in the algorithm: the first iterates over all blocks, the second iterates over all insertion positions. For each inserting position, it builds an O-tree and its corresponding placement. Since O-tree evaluation can be performed in linear time, if there are n blocks to be placed on a chip, the complexity of the deterministic algorithm is $O(n^3)$.

II.3 The Enhanced Perturbing Algorithm

In the deterministic algorithm, there are $2n-1$ insertion positions for each perturbed block. Among them, there exists one position corresponding to a placement minimizing the value of the cost function. It is desirable to determine the best one without constructing all possible new placements at every insertion position. In response to this desire, we present an enhanced perturbing procedure which follows a depth first search order sequence, reducing the time of checking each inserting position to constant time by amortizing the checking of other positions in that sequence. It can achieve the goal of the deterministic algorithm in much less time.

If we visit the nodes of the O-tree shown in Figure II.2 in depth first search order, the numeric labels next to the insertion position indicate the order in which we visit the inserting positions. At each insertion position, we virtually place that deleted

block, and attempt to obtain the value of the cost function without reconstructing the whole placement. In order to make things clear, we restrict the objective function to be the total area of the floorplan in this section. Our algorithm could be extended to minimize a different objective function, such as the weighted sum of the total area and the total wire length. This extension is shown in next section.

II.3.1 Outline of the Enhanced Perturbing Algorithm

Suppose all blocks are to be placed within a chip of size *(H,W)*. *W* denotes the width of the chip, while *H* denotes the height of the chip. We refer to the top part of the chip as the *ceiling*, and the bottom part of the chip as the *floor*. We define *Gap* as the minimum distance of the empty space between the ceiling and the floor in the chip. *Width* is the right boundary of all blocks.

Given an original O-tree, we remove the perturbed block from the O-tree and perform the depth first search for the resulting O-tree. We evaluate all insertion positions during this search. At each possible insertion position, we virtually place the perturbed block and estimate the value of the objective function. In order to prevent the virtually placed block from overlapping with other blocks, we arrange some blocks on the ceiling and some blocks on the floor to make sure that there is enough space for the perturbed block to be virtually inserted. The best result among all possible insertion positions is chosen as the initial for the next iteration.

Procedure *The enhanced perturbing algorithm*

Input: An initial O-tree OT

Output: A reconstructed O-tree OT

for each block u in OT do the following 5 steps:

Step 1: Delete block u from OT so that a resulting O-tree OT_1 *(*T_1*,* π_1*) is obtained*

Step 2: Place all blocks in OT_1 *on the floor.*

Step 3: Slide all blocks up toward to the ceiling.

Step 4: current_block=root

bestcost=infinite

// insert a at the first insertion position

$x_{root}=y_{root}=0$, $x_u=y_u=0$

do step 4.3

index=1

for $T_1[i]$, *i=1,2,... 2n, repeat Step 4.1 through Step 4.3*

Step 4.1: if $T_1[i]$ *is 0*

peel block OT_1->π*[index] down to the floor*

update the contour structure of the ceiling and floor

current_block=OT_1->π*[index]*

index=index+1

else

current_block=parent of current_block

Step 4.2: // virtually place u at position (x_u, y_u)

$x_u=x_{current\_block}+w_{current\_block}$

let ψ *be the set of blocks which are on the floor and whose horizontal spanning intervals overlap with* (x_u, x_u+w_u),

$y_u = max_{k \in \psi} (y_k + h_k)$

Step 4.3: // evaluate this insertion position:

let φ *be the set of blocks which are on the ceiling and whose horizontal spanning intervals overlap with* (x_u, x_u+w_u)

$G_u= min_{k \in \varphi} (y_k -(y_u+h_u))$

newGap=min(Gap, G_u*)*

newWidth=min(Gap, x_u+w_u*)*

newArea=newWidth(H-newGap)*

cost=newArea

if (cost < bestcost)

bestcost=cost

best_insertion_position=this insertion position

Step 5: Insert the deleted block in the best_insertion_position,

construct the corresponding O-tree OT_{new}

OT=min(cost(OT), cost(OT_{new}*))*

II.3.2 An Illustrative Example

A behavioral example is presented in this section to illustrate the enhanced perturbing algorithm.

Suppose the O-tree *OT (001100011101, bcdafe)* shown in Figure II.1 is given as the input O-tree, and block *a* is selected to be perturbed. In step 1, *a* has to be deleted from its original position in *OT*. The resultant O-tree OT_1 *(0011001101, bcdfe)* is shown in Figure II.2. We will traverse OT_1 in depth first search order, and visit the inserting positions in the order listed in Figure II.2.

In step 2 and step 3, the placement of OT_1 is pushed up to the ceiling of the bounding chip. Figure II.3 shows the result of the two steps.

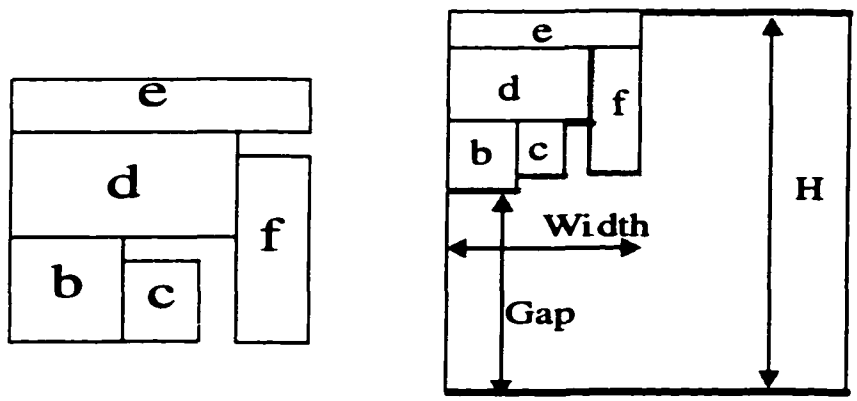


Figure II.3: (a) the placement of OT_1
(b) the placement of OT_1 has been pushed to the ceiling of the bounding box

In step 4, we first virtually insert *a* at the insertion position 1 as shown in Figure II.4. After *a* is inserted, *newGap* is y_b-h_a , *newWidth* is the same as *Width*, and the total area of the placement is *(H-(y_b-h_a))*Width*. We update *bestcost* to be *(H-(y_b-h_a))*Width*, and *best_insertion_position* to be *1*.

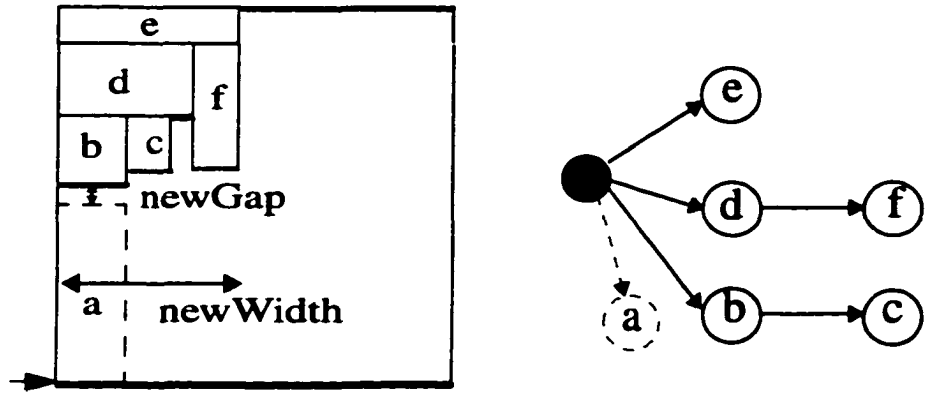


Figure II.4: Insert a at the first insertion position

Then we traverse OT_1 in depth first search order. In iteration $i=1$: $T_1[1]=0$, $\pi_1[1]=b$, we peel block b down to the floor and virtually place a to the right of block b in the insertion position 2 shown in Figure II.5. Therefore the value of the cost function which is also the total area is $(H-(y_c-h_a))*Width$. Since *bestcost* is larger than current *cost*, we update the *bestcost* and the *best_insertion_position*.

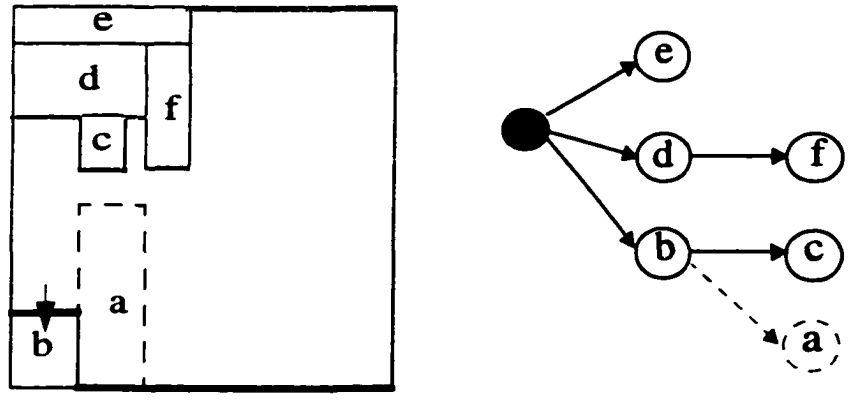


Figure II.5: Insert a in insertion position 2

In iteration $i=2$, $T_1[1]=0$, $\pi_1[2]=c$, we peel c down to the floor. We virtually place a right next to block c in the insertion position 3 shown in Figure II.6. Now the *cost is* $(H-(y_f-h_a))*width$. Since this *cost* is smaller than the *bestcost*, we update the bestcost, and the current best insertion position is position 3.

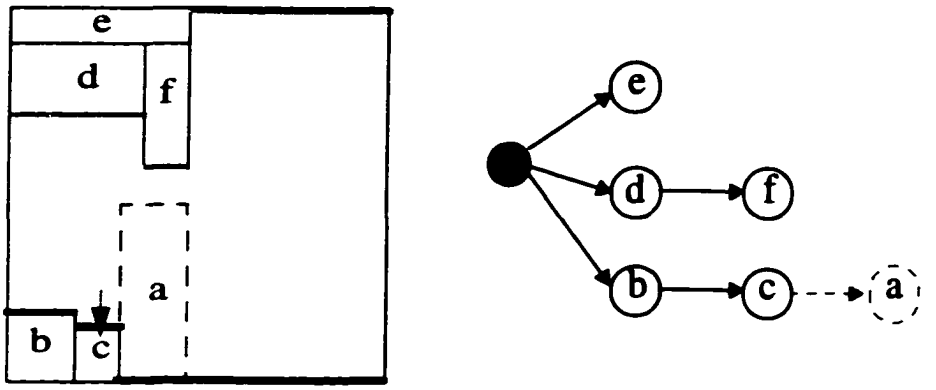


Figure II.6: Insert *a* in insertion position 3

In iteration $i=3$: $T_1[3]=1$, we update the *current_block* to be block *b* which is the parent of block *c*, and virtually place *a* to the right of *b* at the insertion position 4 shown in Figure II.7. Now the *cost* is $(H\text{-}y_d+(h_a+h_c))*Width$. Since *bestcost* is smaller than this *cost*, no update is needed.

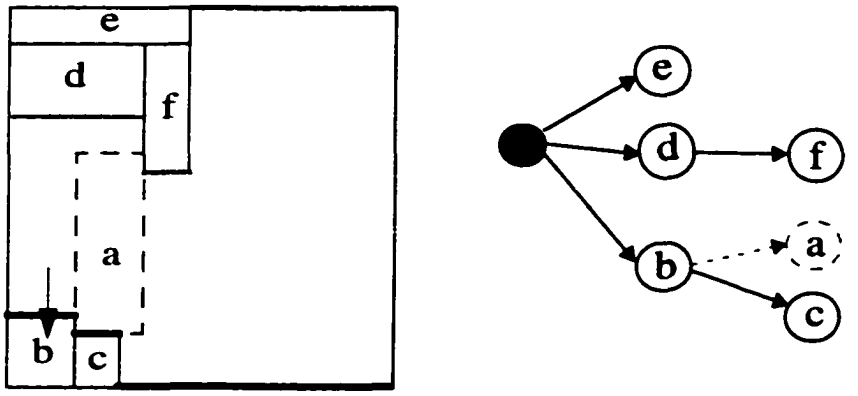


Figure II.7: Insert *a* in insertion position 4

In iteration $i=4$, $T_1[4]=1$, we update the current_block to be the root which represents the left boundary of the bounding chip. We insert block *a* in the fourth insertion position. Now the *cost* is $(H\text{-}y_d+(h_a+h_b))*Width$ which is larger than the *bestcost*, so no update is needed. This operation is shown in Figure II.8.

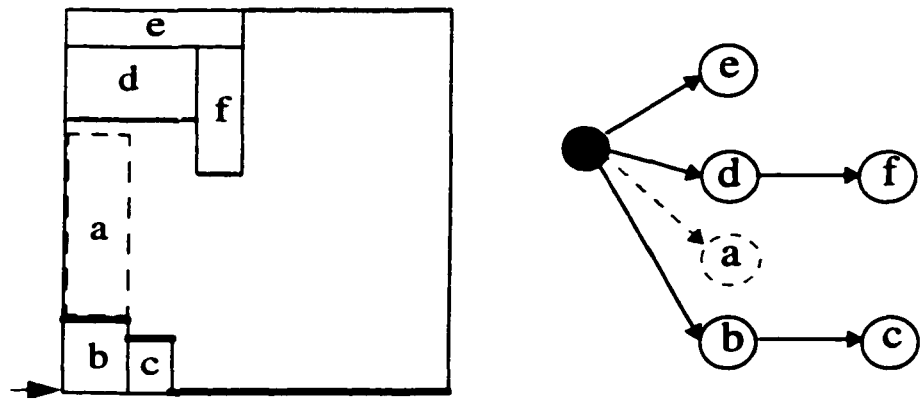


Figure II.8: Insert a in insertion position 5

We continue this process. When the code is '0', we peel down the corresponding block, update the contour structure of both the ceiling and floor, and update the *current_block* to be the block just peeled down. Then we insert block a next to the *current_block*. When the code is '1', we update the *current_block* to be the parent of the previous *current_block*, and then insert block a next to the *current_block*. In both cases, we calculate the new area and update the best cost and the best insertion position when it is necessary. After we traverse all the blocks, we visit all insertion positions, and the best insertion position is the one corresponding to the minimum area. We note that more than one position may lead to the same minimum area. In our example, position 7 is one of them. We insert block a in position 7. The resulting O-tree and its placement are shown in Figure II.9.

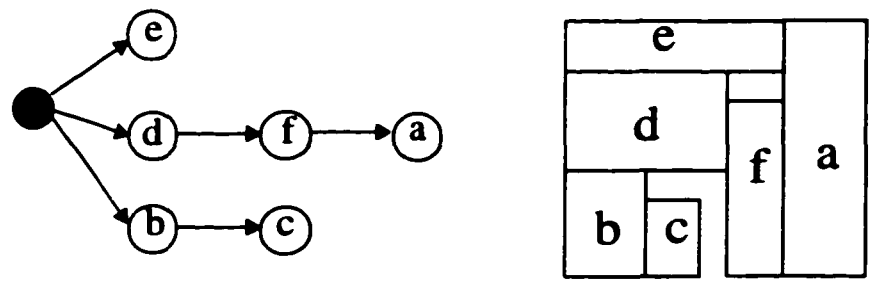


Figure II.9: The resulting O-tree and its placement

After block a is perturbed, the tree is then reconstructed as in Figure II.9. Repeat the procedure until all blocks are perturbed.

II.3.3 The Complexity of the Enhanced Perturbing Algorithm

Theorem 1: *The enhanced perturbing algorithm runs in $O(n^2)$, where n is the number of the blocks of the O-tree.*

Proof: The enhanced perturbing algorithm iterates over all blocks of the given O-tree, so we need to prove that for each iteration the complexity of the algorithm is $O(n)$.

Step 1 takes at most n operations. Steps 2 and 3 can be achieved by the following procedure -- *PushToCeiling*. We will explain that it is $O(n)$ later in this subsection.

In step 4, we visit the nodes in depth first search order, so each node is visited exactly twice: once at the nodes encoded '*0*' and once at the nodes encoded '*1*'. This loop executes exactly $2n$ times. Inside the loop, we perform four major operations:

(1) peel down a block from the ceiling

(2) find the local maximum of y-coordinate of the floor

(3) find the local minimum of y-coordinate of the ceiling

(4) update the contours of the ceiling and the floor

We only need to pass a limited set of blocks on the above four operations. The number of the blocks accessed is equal to the number of edges inserted in the vertical constraint graph. The constraint graph is planar, and the number of edges in the vertical constraint graph is $O(n)$. In the whole loop, we go through every edge in the vertical constraint graph exactly twice. The overall complexity for step 4 is then linear.

Therefore the complexity for the enhanced perturbing algorithm is $O(n^2)$.

We apply the following algorithm *PushToCeiling* to step 3. The basic idea of this algorithm is to transform the given O-tree to another O-tree structure whose siblings are in the reverse order of the original O-tree, and then place the new structured O-tree on the ceiling.

```
Procedure PushToCeiling
Input: O-tree OT_d
Output: the vertical constraint graph and the placement of blocks in OT_d on the ceiling.
 perm=1
 current=head
 for i=1 to 2n
    new_T[2n+1-i]=1-T[i]  // put the siblings in reverse order
    if (T[i]==0)
       put the π[perm] after the current block
       perm=perm+1
    else
       current=prev(current)
//write new permutation in the correct corresponding order
 current=head
 for i=1 to n
    current=next(current)
    new_π[i]=current
place O-tree(new_T, new_π) on the ceiling
```

Theorem 2: *Algorithm PushToCeiling runs in linear time*

This follows since converting an O-tree to another O-tree with reverse order sibling structure takes linear time, and transforming an O-tree to the placement takes linear time.

II.4 Extended Cost Function

In physical design, wires are usually considered in addition to the area in the objective function. We extend our objective function to be the weighted sum of the area and wire length in this section. Wire length is defined by the half perimeter of the minimum bounding box.

At each insertion position, the area of the corresponding placement can be

obtained by the method described in the above section. In order to save the computational cost, we only consider the nets related to the inserted block for the wire length. We use the same notation here as in the procedure for the enhanced perturbing algorithm. OT_1 is the resulting O-tree obtained by removing block u from the O-tree OT. At insertion position i, the block u is virtually placed in (x_u, y_u). The wire length of the corresponding placement can be approximated by the following steps:

for each pin_i of block u

$x=x_u$+relative x-coordinate of pin_i

$y=y_u$+relative y-coordinate of pin_i

net_i is the net that pin_i need to be connected

(x_1, x_2, y_1, y_2) is the boundary of net_i without block u in the placement of OT_1

if $x<x_1$ $x_1=x$

if $x>x_2$ $x_2=x$

if $y<y_1$ $y_1=y$

if $y>y_2$ $y_2=y$

change_wire=the wire length changed by the above loop

newWirelength=change_wire+wirelength(OT_1)

*cost=w_1*newArea+w_2*newWirelength*

II.5 Complete Design of a Floorplan

In addition to improving an existing floorplan, the idea of the enhanced perturbing algorithm can be used to carry out the complete design of a floorplan. Starting with a random permutation of the labels of the blocks, we can heuristically construct a floorplan in the following way:

Procedure *Complete floorplan design*

Input: A random sequence from 1 to n

Output: A floorplan of block 1 to block n

construct an O-tree OT with empty nodes

for $\pi[1]$ to $\pi[n]$

evaluate all possible inserting position in OT

construct a new O-tree new_OT by inserting π[i] in the best insertion position

OT=new_OT

run the enhanced perturbing algorithm to OT

output the placement of OT

By using the order of the given random sequence, we insert the blocks to the constructed tree in the best position each time. We then improve the already constructed O-tree by running the enhanced perturbing algorithm. At each step we perform a greedy strategy. This process continues until a complete pass through all the blocks produces no improvement.

In order to avoid getting stuck at a local optimum, we generate a large amount of random sequences. For each sequence, we obtain a placement using the above algorithm, and choose the best one according to the cost function.

Flexibility of the Blocks

If a block is a soft block which has several weight and height tuples, we evaluate all possible configurations at each insertion position and choose the best configuration. For example, as shown in Figure II.10, the inserted block a has two weight and height tuples . At insertion position 1, we test both dimensions of block a, and choose the best dimension according to the cost function.

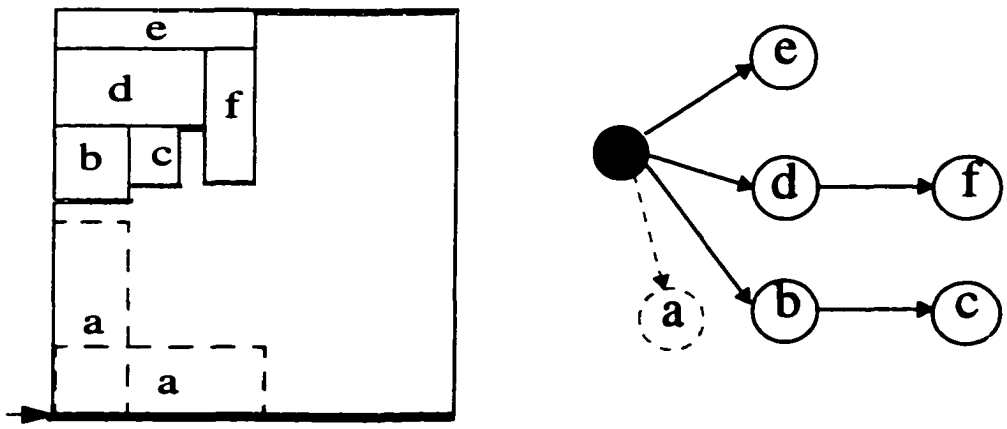


Figure II.10: Multiple dimensions for a inserted block

II.6 Experimental Results

We have implemented our algorithm in C on a Sun Ultra60 workstation. The test cases are the five MCNC benchmarks. We compared our algorithm with the deterministic algorithm. The experimental results are reported in Tables II.1, II.2, and II.3. All the results have been run on the same Sun workstation. The cost function we use here is w_1*area+w_2*wirelength. Tables II.1, II.2 and II.3 correspond to the following sets of $\{w_1, w_2\}$ values: {0, 1}, {1, 0} and {0.5, 0.5}. DA denotes the deterministic algorithm and EPA denotes our algorithm.

The initial sequences of the problems in the tables are randomly generated. In each test case, we use 100 randomized sequences and present the best results among the 100 runs. The running time of our algorithm on each of the test examples is substantially less than that of the deterministic algorithm. For circuit *apte* (which has only 9 cells and is the smallest circuit among the MCNC benchmarks), our algorithm is about 3 times faster than the deterministic algorithm under all three different cost functions. But for the largest circuit *ami49* with 49 cells, our algorithm is about 20 times faster on average. The larger the circuit, the greater the performance gain with our algorithm.

Table II.1 is a summary of results for the case where the cost function is the total area. Among the five circuits, our algorithm has slightly better solutions than the deterministic algorithm in three of them. In the other two test examples, the areas of the placements by our algorithm are no more 0.55% larger than those by the deterministic algorithm. Therefore the two algorithms produce equivalent results.

circuit	w_1=1 w_2 =0			
	time(*min*)		area(mm^2)	
	EFA	DA	EPA	DA
apte	0.19	0.63	46.92	47.1
xerox	0.63	1.97	20.21	20.1
hp	0.32	0.95	9.159	9.21
ami33	1.98	23.83	1.242	1.25
ami49	6.76	123.5	37.73	37.6

Table II.1: Area results for MCNC benchmarks

circuit	w_1=0 w_2 =1			
	time(*min*)		wire(*mm*)	
	EPA	DA	EPA	DA
apte	0.25	0.79	316.8	317
xerox	0.65	2.66	372.2	368
hp	0.32	1.50	150.4	153
ami33	2.95	37.52	51.58	51.5
ami49	11.46	235.2	629.3	636

Table II.2: Wire length results for MCNC benchmarks

circuit	w_1=0.5 w_2 =0.5					
	time(*min*)		area(mm^2)		wire(*mm*)	
	EPA	DA	EPA	DA	EPA	DA
apte	0.24	0.77	51.92	51.92	320.7	320.7
xerox	0.68	2.37	20.42	20.42	380.6	380.6
hp	0.35	1.40	9.384	9.490	151.9	152.6
ami33	3.42	39.15	1.299	1.283	52.13	51.31
ami49	11.67	255.3	39.92	39.55	702.8	688.7

Table II.3: Area and wire length for MCNC benchmarks

In Tables II.2 and II.3, wire length has been considered in the cost function. Comparing with the deterministic algorithm, we have a -2% to 2% improvement in area and in wire length. On average, the two algorithms generate the same high quality placements.

For the cost function with weights 0.5 for both the area and wire length, we display five MCNC benchmark placements. Figure II.11, II.12 and II.13 show that the placements of three small circuits *apte*, *xerox* and *hp* are all obtained in less than 1 minute. The placement of *ami33* is presented in Figure II.14. This circuit has 33 cells, and the placement takes only 3.42 minutes. Figure II.15 shows the largest circuit *ami49* of the five examples. This placement was performed in about 12 minutes.

The above results clearly show that our algorithm obtains results as good as that of the deterministic algorithm, while taking much less time. Therefore, our algorithm appears to be more effective.

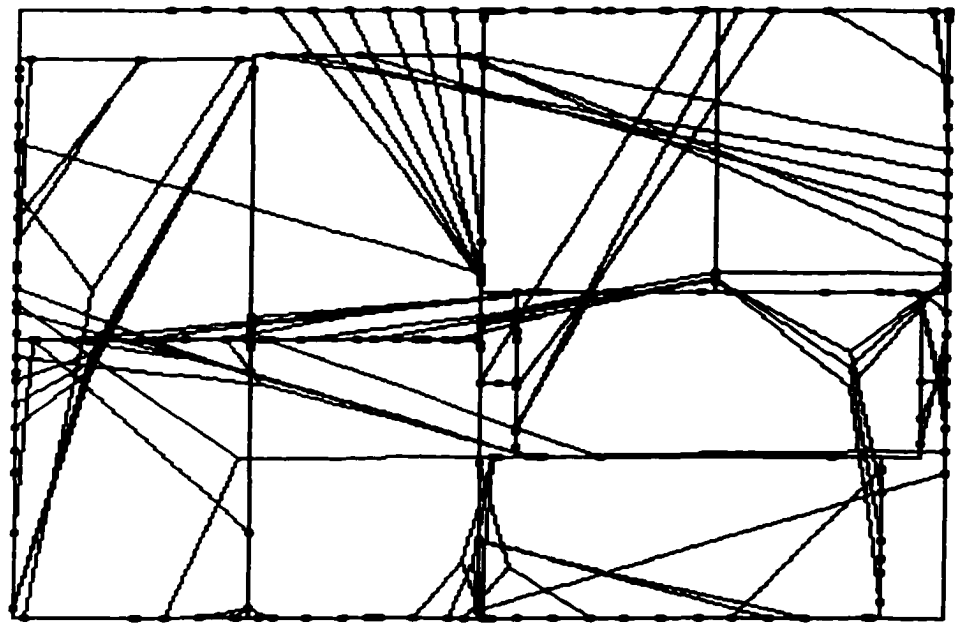


Figure II.11: Circuit apte

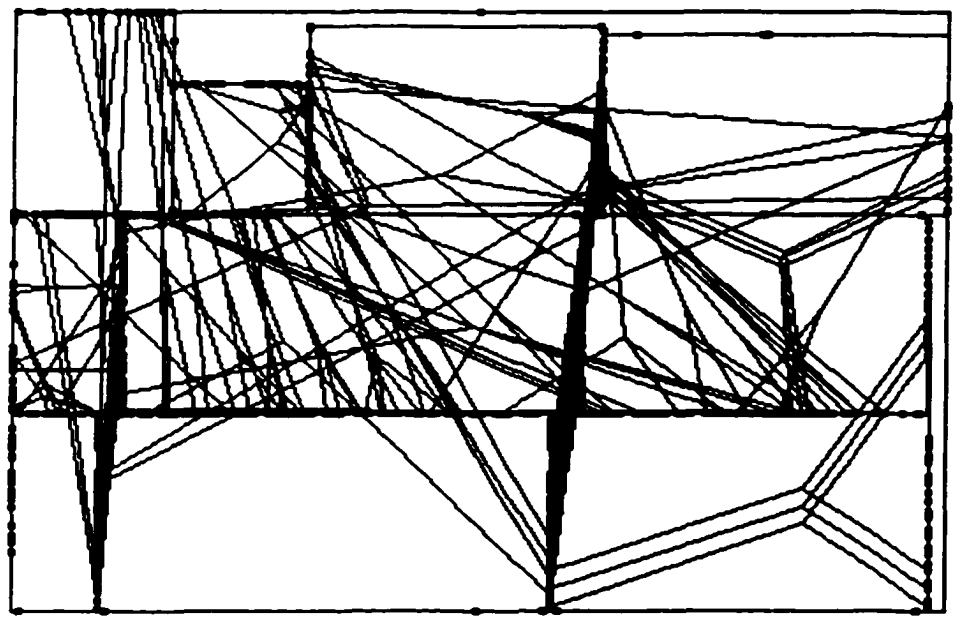


Figure II.12: Circuit xerox

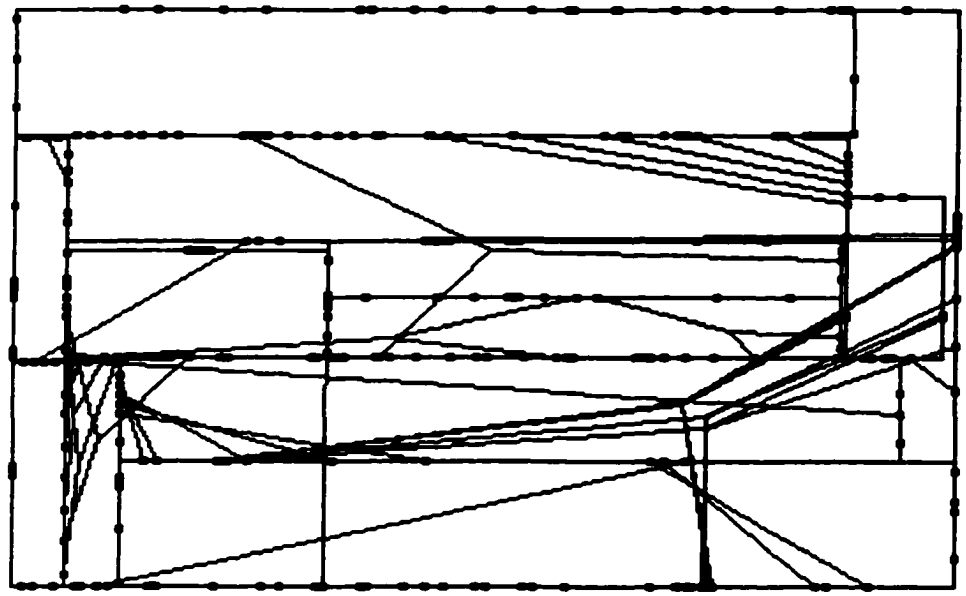


Figure II.13: Circuit hp

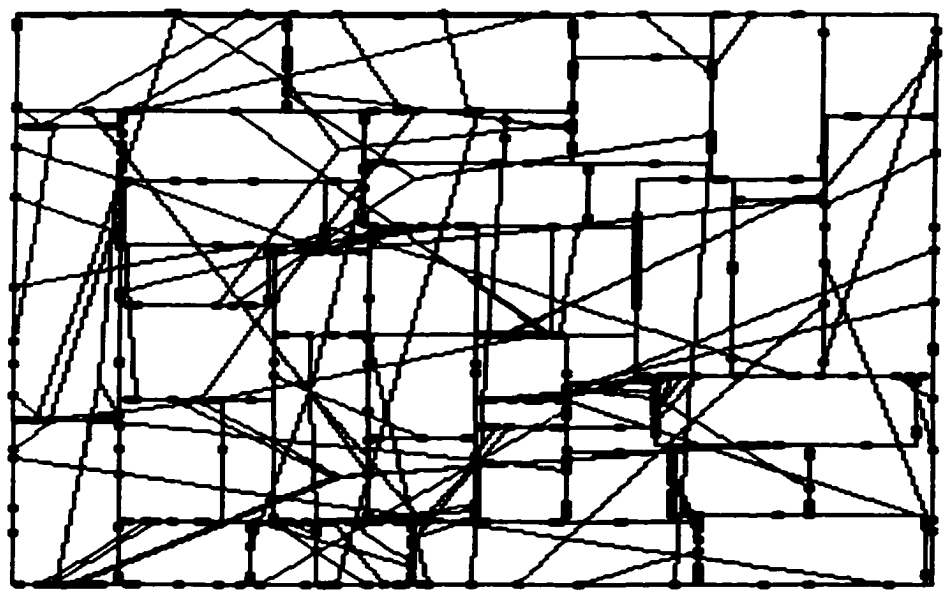


Figure II.14: Circuit ami33

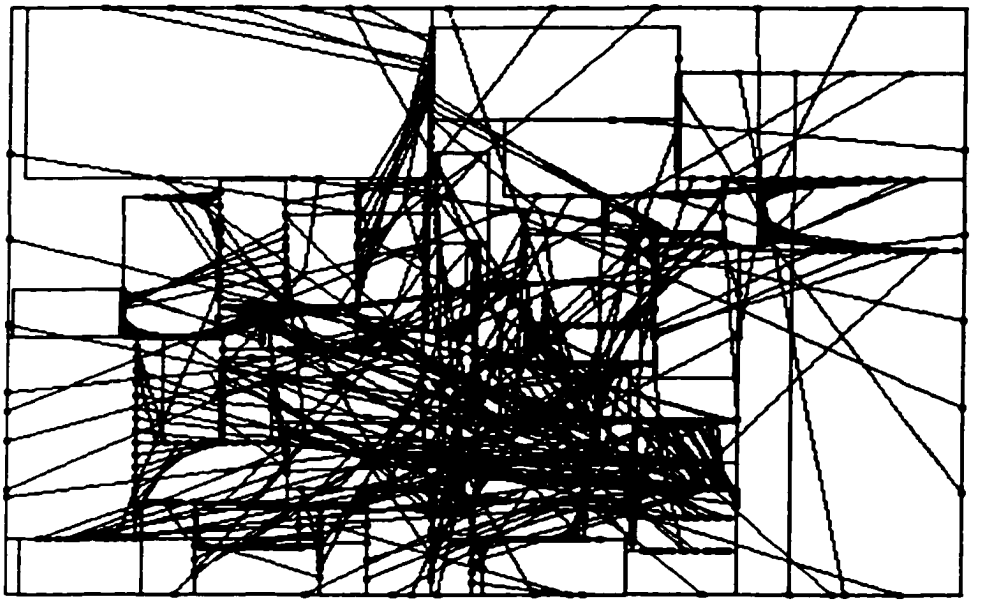


Figure II.15: Circuit ami49

II.16 Summary

We have proposed a new method utilizing the properties of an O-tree to improve an existing floorplan or carry out a complete design of a floorplan in faster time. Since the speed up is proportional to the size of the problem, it is especially useful for handling large circuits. Our algorithm can cover more random initial sequences in the same amount of time, and potentially obtain better solutions.

Portions of this chapter is a reprint of the material as it appears in "An Enhanced perturbing Algorithm for Floorplan Design Using the O-tree Representation", Yingxin Pang; Chung-Kuan Cheng; Takeshi Yoshimura, Proceedigns of international symposium on Physical Design, 2000. The dissertation author was the primary investigator and single author of this paper.

Chapter III: Block Placement with Symmetry Constraints

III.1 Introduction

The ability of placement tools to optimize complex layout-related objectives, while having the flexibility to handle a large variety of specific constraints, is crucial in order to automatically produce high-quality layouts in terms of density and electrical performance. In device-level analog placement, dealing with symmetry constraints is essential as analog circuits use very often differential architectures based on electrically symmetric networks. Symmetry is widely used in analog layouts to match interconnection parasitics and device parameters, or to balance thermal effects.

The issue of symmetry has been addressed so far in the context of two distinct classes of analog placement solutions. The first class employs the absolute (or flat) representation of placement configurations, where cells are specified in terms of absolute coordinates on a gridless plane. Symmetry constraints, requiring groups of devices to be placed symmetrically with respect to one or several axes, are easily modeled by means of algebraic relations between cell coordinates [4],[10][14-15].

This first class of tools, exploring absolute placement configurations with simulated annealing algorithms, has proven to be successful when dealing with industrial examples (e.g., KOAN/ANAGRAM II [4], PUPPY-A [14,15]). However, they may converge slowly, due to the huge size of the search space which may contain also infeasible placement configurations (as cells are allowed to overlap). In addition, the tuning of these tools is usually difficult, requiring a significant amount of testing effort.

The second class of solutions employs topological representations of placement configurations, where cell positions are specified based on encoded topological relations. In the ILAC system [23], symmetry is handled in the context of a slicing floorplan model [19]. Although slicing representations are able to preserve hierarchy, the resulting layouts may be poor in terms of area as slicing structures cannot cover all the topological relationships and, therefore, all the possible placement configurations.

More recently, several non-slicing topological representations have been proposed (the sequence-pair [16], the bounded sliceline grid (BSG) [18], the ordered tree (O-tree) [7]), which can be used to explore only the set of feasible placement configurations. Symmetry constraints can be handled within the sequence-pair representation as shown in [1].

In this chapter, we present a novel placement technique based on the O-tree representation, in the presence of positioning and symmetry constraints. The good performance of our placement tool when dealing with several analog designs taken from industry proves the effectiveness of our approach.

The rest of the chapter is organized as follows: Section III.2 introduces the placement problem with symmetry constraints. Section III.3 thoroughly presents the theoretical framework for solving the rectangle packing problem with symmetry constraints in the context of the O-tree representation. Section III.4 gives an overview of our experimental results, and Section III.5 presents a basic summary of our research.

III.2 Symmetry Constraints

Symmetry constraints can be formulated in terms of *symmetry pairs, self-symmetric devices,* and *symmetry groups. A symmetry pair* is a couple of blocks having the same dimensions, which have to be placed symmetrically with respect to an axis. A self -symmetric device is a device whose center is placed on a symmetry axis. A *symmetry group* is a set of symmetry pairs which share a common axis. Assuming the common axes are horizontal in all symmetry groups, then the symmetry constraints can be formulated as follows: if $\{(a_1, b_1), (a_2, b_2), \ldots, (a_k, b_k)\}$ is a symmetry group, and (a_i, b_i) is a symmetry pair, for $i=1, \ldots, k$ then

$$x_{a_i} = x_{b_i} \tag{III•1}$$

$$y_{a_i} + y_{b_i} + h_{a_i} = 2y_s \tag{III•2}$$

where x_{a_i}, x_{b_i} are the x-coordinates of blocks a_i and b_i, y_{a_i}, y_{b_i} are the y-coor-

dinates of blocks a_i and b_i, y_s defines the position of the common symmetry axis.

The first type equalities are called *horizontal symmetric constraints*; the second type are *vertical symmetric constraints.*

Given an O-tree and a symmetry group, the minimum area placement - obtained as described in chapter I - does not usually satisfy the symmetry constraints. For instance, assuming that the pairs of blocks *((b,h),(e,f))* constitute a symmetry group, the placement in Figure III.1(b) corresponding to the O-tree in Figure III.1(a) does not exhibit the required symmetry properties. At a first glance, the solution may seem straightforward: in addition to the positioning constraints, the placement procedure should also take into account the symmetric constraints (III.1) and (III.2), and push the block left and right, up and down. Unfortunately, this does not always work as the added symmetric constraints may conflict with the O-tree positioning constraints, as it happens if the given O-tree looks like the ones in Figure III.3(a) or III.3(b). Therefore, only the O-trees having the structure consistent to the symmetry constraints need to be considered during the search space exploration.

The next section thoroughly studies the problem of rectangle packing with symmetry constraints (RPSC).

III.3 Rectangle Packing with Symmetry Constraints

III.3.1 Symmetric X-feasible O-trees

If an O-tree can lead to a placement which satisfies both the horizontal positioning constraints and the horizontal symmetric constraints (III.1), then the O-tree is called *symmetric x-feasible.*

In order to determine whether an O-tree is symmetric x-feasible or not, horizontal constraint graph G_x is built. This directed graph has the same nodes as the given O-tree, and the same directed edges as horizontal positioning constraints. In addition, for each symmetric pair of blocks *(a,b)*, two new arcs *(a,b)* and *(b,a)* are introduced (unless positioning constraint paths between *a* and *b* already exist).

Example: Assuming the pairs of blocks *(b,h)* and *(e,f)* are symmetric, the hori-

zontal constraint graph G_x of the O-tree displayed in Figure III.1 (a) can be visualized in Figure III.2 (a), where the broken arcs model the horizontal symmetric constraints (1). The edges of G_x are weighted: if the edge *(u,v)* corresponds to a positioning constraint, its weight is $w(u,v)=w_u$, the width of the block represented by node u; if the edge corresponds to a symmetric constraint, then $w(u,v)=0$.

The existence of positive cycles in the directed graph G_x determines the symmetric x-feasibility of the O-tree.

Theorem 1: *If the horizontal constraint graph G_x does not contain positive cycles, then the corresponding O-tree is symmetric x-feasible. In addition, one can build a minimum width placement satisfying both horizontal positioning and symmetric constraints in $O(n^2)$ time.*

Proof: In a minimum width placement the x-coordinate of each block must be equal to the longest path length from the "root" node to its corresponding node in G_x. Finding a minimum width placement is equivalent to be a single-source longest path problem, which can be solved with the Bellman-Ford algorithm [5]. If G_x contains no positive cycles, the longest path problem is provably consistent and the Bellman-Ford algorithm converges.

As the x-coordinates of blocks are longest path lengths, for any directed edge *(u,v)* in G_x, we have

$$x_v \geq x_u + w(u, v)$$

the horizontal positioning constraints being satisfied. We claim that the horizontal symmetric constraints (III.1) are satisfied as well.

Let *(a,b)* be a symmetry pair and assume x_a is not equal to x_b. To be specific, assume $x_b > x_a$. By the construction of G_x, either *(b,a)* is a directed edge of *0 weight*, or there is a path from b to a in the O-tree and, therefore, in G_x. If *(b,a)* is an edge of *0* weight then, due to the longest path definition,

$$x_a \geq x_b + w(b, a) = x_b$$

our assumption is contradicted. If there is a path from b to a in the original O-tree, the path length is positive and, therefore, $x_a > x_b$, which also leads to a contradiction. It fol-

lows that the horizontal symmetric constraints (III.1) are satisfied for any symmetric pair and, hence, the O-tree is symmetric x-feasible.

The complexity of the Bellman-Ford algorithm is $O(VE)$, where V is the number of nodes and E is the number of edges [5]. Since the O-tree has n edges, the number of edges in G_x is $O(n)$. Taking also into account that G_x has $n+1$ nodes, the complexity results to be $O(n^2)$.

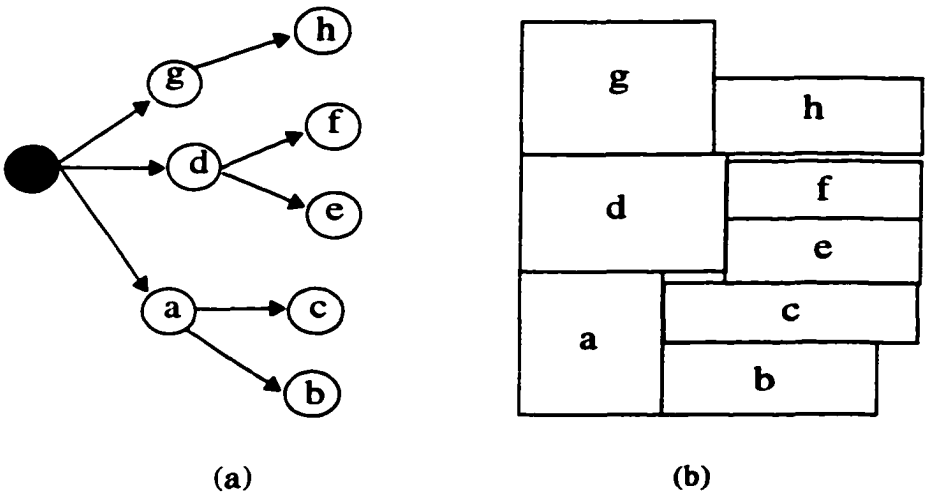


Figure III.1: An O-tree and its placement

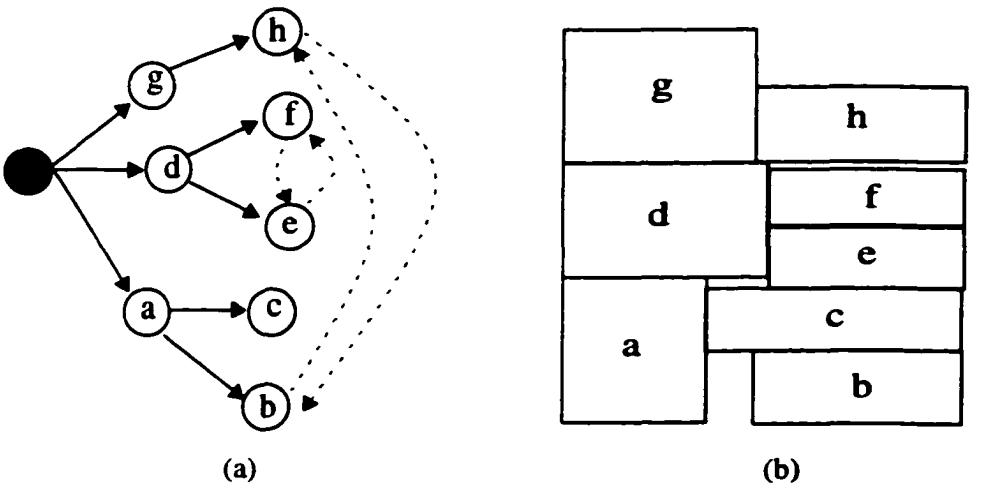


Figure III.2: A symmetric x-feasible O-tree and its resulting placement

Figure III.2 (a) shows an example of *symmetric x-feasible* O-tree, the symmetry group being *{(b,h), (e,f)}*. Figure III.2 (b) displays the minimum width placement derived form the O-tree in Figure III.2 (a).

Theorem 2: *If there is a positive cycle in G_x, then the O-tree is not symmetric x-feasible.*

Proof: If G_x contains a positive cycle, this cycle must include at least one newly added edge between nodes corresponding to symmetric blocks. (The other edges are derived directly from the O-tree, which is obviously acyclic.) If *(a,b)* is such a directed edge of zero weight, it follows that $x_a > x_b$, as there is a path of positive length from b to a formed by all the edges of the given positive cycle, excepting *(a,b)*. It follows that the O-tree is not symmetric x-feasible and the horizontal symmetric constraints conflict with the O-tree constraints.

Theorems 1 and 2 prove the equivalence between the existence of positive cycles in the horizontal constraint graph G_x and the O-tree property of symmetric x-feasibility. The graph G_x will contain positive cycles when:

Case 1: *(c,d)* is a symmetry pair and node c is an ancestor (not necessarily direct) of node d, as shown in Figure III.3 (a) where the pair *(e,f)* creates a positive cycle;

Case 2: there is a sequence of q *(>1)* symmetry pairs (c_1, d_1), (c_2, d_2), ... , (c_q, d_q), such that c_1 is an ancestor of d_2, c_2 is ancestor of d_3, ... , c_q is an ancestor of d_1. In Figure III.3 (b), the pairs *(b,h)* and *(e,f)* create such a positive cycle.

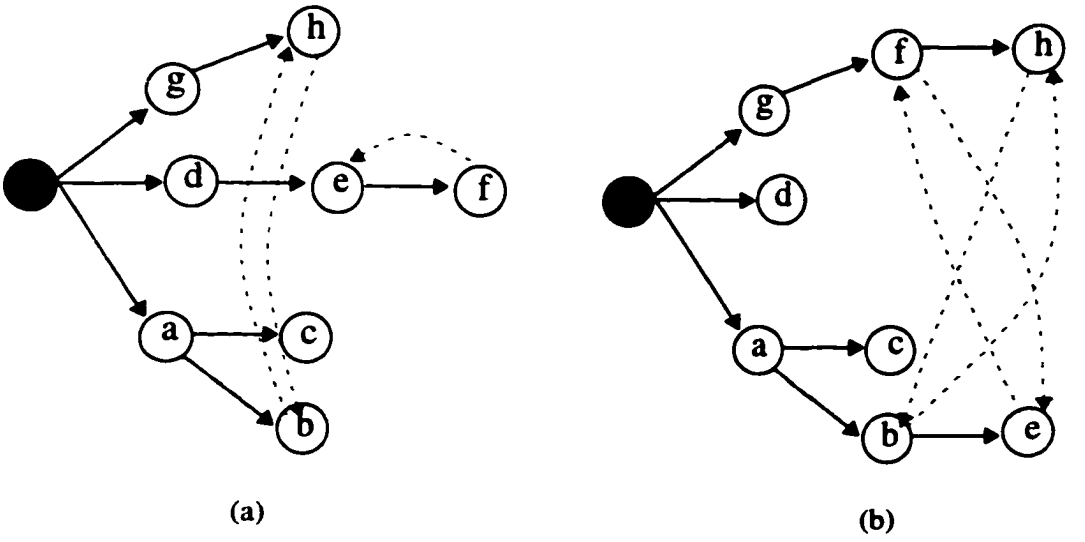


Figure III.3: Examples of O-trees which are not symmetric x-feasible

Remark: The proof of Theorem 1 relies on the Bellman-Ford algorithm [5], having the complexity $O(n^2)$ in our case, to construct the minimum width placement or to detect the existence of positive cycles in graph G_x. In practice, however, it is possible to exploit the special structure of the O-tree and perform the same tasks in linear time: the basic idea is to utilize a breadth-first topological sort, gradually coalescing the pairs of nodes corresponding to the symmetry pairs. When the resulting graph is acyclic, the algorithm has the complexity $O(V+E)$, that is $O(n)$ in our case; otherwise, the algorithm stops even earlier. The use of Bellman-Ford algorithm in the proof of Theorem 1 is due only for reasons of clarity.

III.3.2 Symmetric Y-feasible O-trees

If an O-tree can lead to a placement which satisfies the vertical symmetric constraints (III.2) and, at the same time, which does not violate the vertical positioning constraints, then the O-tree is called *symmetric y-feasible*.

In order to determine whether an O-tree is symmetric y-feasible or not, we construct a vertical constraint graph G_y. This directed graph has the same nodes as the given O-tree. Part of the directed edges represent the positioning vertical constraints

of the O-tree. These edges can be determined, similarly to the original placement construction, as follows:

for each block B_i , $i=1, ..., n$, let $\psi(i)$ be the set of block indexes k lower than i in permutation π , whose spanning intervals $(x_k, x_k + w_k)$ overlap $(x_i, x_i + w_i)$; if $\psi(i)$ is non-empty, for each $k \in \psi(i)$ introduce in graph G_y a directed edge (B_k, B_i) , unless there is no other edge (B_k, B_j) , with $j \in \psi(i)$ (in order to disregard unnecessary transitive arcs). In order to handle the vertical symmetric constraints (III.2), for each edge (B_k, B_i), we define the weight as $(h_k+h_i)/2$. In this way, the nodes will correspond to the center of the blocks rather than the bottom of the blocks. Additional edges modeling the vertical symmetric constraints (III.2) must be added in G_y as well. Assuming that all the symmetry pairs (a,b) have their elements in topological order relative to the positioning constraints, the additional edges are introduced as follows:

for every symmetry group $\{(a_1, b_1), (a_2, b_2), ..., (a_k, b_k)\}$
for every two symmetry pairs (a_i, b_i), (a_j, b_j)
if there is a path from a_i to a_j then
    add to G_y a directed edge (b_j, b_i), unless the edge does exist already;
    let the weight of the edge (b_j, b_i) be equal to the longest path length from a_i to a_j;
if there is a path from b_i to b_j then
    add to G_y a directed edge (a_j, a_i), unless the edge does exist already;
    let the weight of the edge (a_j, a_i) be equal to the longest path length from b_i to b_j;

Example: Assuming the pairs of blocks (b,h) and (e,f) are symmetric, the vertical constraint graph G_y of the O-tree displayed in Figure III.1 (a) can be visualized in Figure III.4 (a). The plain arcs represent the O-tree vertical positioning constraints. The broken arc (b,e) was introduced during the execution of the procedure shown above, as there was a path from node f to node h . This arc was added to model the vertical symmetry constraints, as it will be explained in the proof of Theorem 3 .

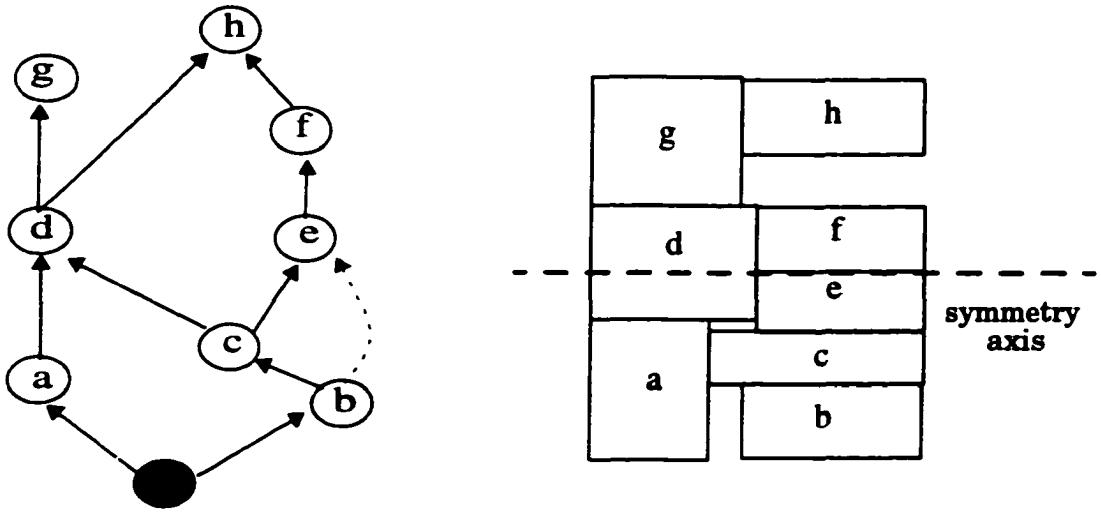


Figure III.4: A symmetric y-feasible O-tree and its resulting block placement

The existence of cycle in the directed graph G_y determines the symmetric y-feasibility of the O-tree.

Theorem 3: *If the vertical constraint graph G_y does not contain cycles, then the corresponding O-tree is symmetric y-feasible. In addition, one can build a minimum height placement satisfying both vertical positioning and symmetric constraints in $O(n^2)$ time.*

Proof: Denoting (x_i , y_i) the coordinates of the left-bottom corner of block B_i, $z_i=y_i+h_i / 2$ the y-position of the center of B_i, and given a symmetry group $U=\{(a_1, b_1), (a_2, b_2), \ldots , (a_k, b_k)\}$, a placement which has the properties stated in Theorem 3 can be constructed as follows:

(1) let $z_{root}=0$

execute the Bellman-Ford single-source longest path algorithm in graph G_y (considering the root node as the source),

let z_i = the longest path length from root to node B_i .

(2) determine the position of the common symmetry axis

$$y_s = \max_{(a_i, b_i) \in U} \left\{ \frac{z_{a_i} + z_{b_i}}{2} \right\}$$

(3) execute a topological sort of the nodes in G_y,

reorder the symmetry pairs (a_i, b_i) in U such that nodes b_i are ordered according to the topological sort

(4) for each symmetry pair (a_i, b_i) in U

$$d = y_s - \frac{z_{a_i} + z_{b_i}}{2}$$

if $d > 0$ then

$$z_{b_i} = z_{b_i} + 2d$$

execute the Bellman-Ford single-source longest path algorithm, considering node b_i as the source

// as a result, the values z_j are updated for all the nodes j descendents of b_i;

// this is equivalent to pushing upwards all the blocks j on top of b_i

Since the y-positions of the blocks are adjusted (step 4) according to the topological order, the y-positions of the early symmetry pairs are not affected by the adjustments involving symmetry pairs processed later. After each iteration in step 4, the current pair (a_i, b_i) will get y-coordinates of block centers symmetric relative to the position y_s of the common axis. One by one, all vertical symmetric constraints are satisfied. Taking into account that the vertical constraint graph G_y was built such that the positioning constraints are inherently satisfied, the O-tree is symmetric y-feasible. As the position y_s of the symmetry axis has the minimum possible value, the final placement has a minimum height.

Steps 1 and 4 contain the most expensive operations. Due to the Bellman-Ford algorithm, step 1 requires $O(n^2)$ time. Step 4 has k iterations; in each iteration the single-source longest path algorithm may be executed. Apparently, the worst-case complexity of step 4 is cubic. Actually, each edge in G_y is examined no more than once and,

therefore, the complexity is $O(n^2)$. In conclusion, the algorithm described above runs in $O(n^2)$ time.

Remark: Due to the fact that positioning constraints can be algebraically modeled as linear inequalities, and the symmetric constraints (III.2) are linear equalities, the problem of finding the minimum height placement can be expressed as a linear programming problem having as cost function the height of the chip.

Figure III.5 shows an example where the O-tree is not symmetric y-feasible. The graph G_y in Figure III.5 (a) corresponds to the same O-tree in Figure III.1 (a) with nodes *f* and *h* interchanged, and assuming the same symmetry group *{(b,h), (e,f)}* as in our previous examples. The placement corresponding to the positioning constraints (Figure III.5 (b)) does not satisfy the vertical symmetric constraints. The main reason is the presence of cycles in the vertical constraint graph G_y.

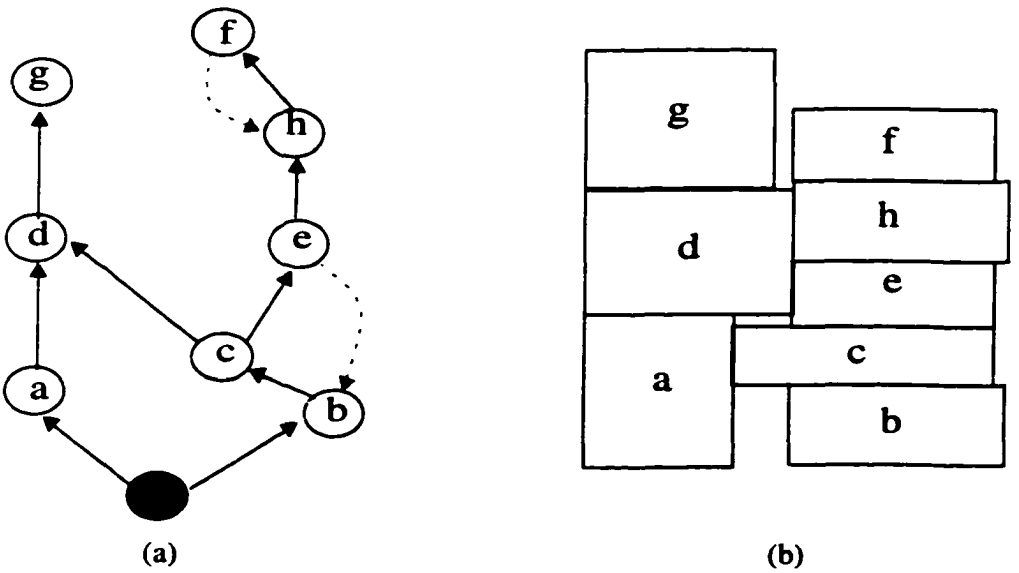


Figure III.5: An example of O-tree which is not symmetric y-feasible

Theorem 4: *If there is a cycle in G_y, then the O-tree is not symmetric y-feasible.*

Proof: For every symmetry pair *(a,b)*, as we may assume $x_a = x_b$, there is a path between the nodes *a* and *b* containing only edges corresponding to positioning con-

straints. Let *(a,b)* and *(c,d)* be two symmetry pairs belonging to the same group, and suppose there is such a path from a to b and another one from c to d. A cycle in G_y must contain a "broken" arc (corresponding to a vertical symmetric constraint), as the positioning constraints determine an acyclic graph. Let *(d,b)* be such a "broken" arc contained in a cycle: it follows there is a path from node b to node d in G_y and, consequently, $z_d > z_b$. But this "broken" arc *(d,b)* could be introduced in G_y only due to the existence of a path from node a to node c. This implies $z_c > z_a$, which together with the other inequality $z_d > z_b$, implies that the symmetry pairs cannot have a common symmetry axis (as $(z_c+z_d)/2 > (z_a+z_b)/2$). This contradicts the symmetry group assumption.

III.3.3 Symmetric Feasible O-trees

An O-tree is called *symmetric feasible* if it is both symmetric x- and y- feasible. Given n rectangular blocks, a set of symmetric constraints, and a n-node symmetric feasible O-tree, one can build a minimum area placement as described in subsections 3.1 and 3.2, where the positions of the blocks satisfy both the O-tree constraints and the given symmetric constraints. On the other hand, there is a reciprocal property:

Theorem 5: A minimum area rectangle packing with symmetric constraints can be represented by a symmetric feasible O-tree.

Proof: In [7] it was proven that a minimum area rectangle packing can be represented by an O-tree. We shall first prove that this O-tree is symmetric x-feasible. Let *(a,b)* and *(c,d)* be two symmetry pairs belonging to the same symmetry group. As the spanning intervals (x_a, x_a+w_a) and (x_b, x_b+w_b) are identical (it was assumed that symmetry axes are horizontal), there are no horizontal positioning constraints between blocks a and b : node a cannot be the ancestor of node b and vice versa. Therefore, the (broken) arc *(a,b)* cannot create a positive cycle in the horizontal constraint graph G_x (see Case 1, subsection 3.1). If the spanning intervals (x_a, x_a+w_a) and (x_c, x_c+w_c) of the two pairs do overlap, then the nodes a, b, c, d are on different branches of graph G_x. If the spanning intervals do not overlap, then we may have, for instance,

$$x_a + w_a \leq x_c$$

node a is an ancestor of c in the O-tree; then node d cannot be an ancestor of b, as it

would follow:

$$x_c + w_c = x_d + w_d \leq x_b = x_a \leq x_c - w_c$$

this is absurd. Both situations show that Case 2 (subsection 3.1) cannot happen. Therefore, the O-tree is symmetric x-feasible.

In a similar way, one can prove the property of symmetric y-feasibility. If the spanning intervals of the two pairs *(a,b)* and *(c,d)* do not overlap, then nodes *a*, *b* and, respectively, *c*, *d* lie on different branches of the vertical constraint graph G_y. As there is no path from, e.g., node *a* to node *c* , no new (broken) arc *(d,b)* can be created (see subsection 3.2) and, therefore, the two pairs cannot produce any cycle in graph G_y. If the two spanning intervals do overlap, then nodes *a,b,c* and *d* are on a same path in G_y . As the two pairs have the same symmetry axis, the nodes corresponding to one pair will be in between the other two nodes on the respective path. The two (broken) arcs *(a,c)* and *(d,b)* introduced, as shown in subsection 3.2 , during the construction of graph G_y , do not create any cycle in this graph.

Theorem 5 shows that the solutions of the rectangle packing with symmetry constraints (RPSC) problem can be represented only by symmetric feasible O-trees.

Our placement tool employs the simulated annealing algorithm as the exploration engine for the O-tree search space. According to theorem 5, only the symmetric feasible O-trees are taken into account, as they lead to placement configurations satisfying the symmetric constraints. The test of symmetric feasibility and the placement construction (described in Sections 3.1 and 3.2) are done simultaneously, according to the following general scheme:

Inputs: (i) the set of rectangular blocks, (ii) the set of symmetric constraints,
(iii) an O-tree generated by the simulated annealing
construct the horizontal constraint graph G_x*;*
if graph G_x *is symmetric x-feasible*
/ / if G_x *has no positive cycles*
compute the x-coordinates of the blocks;
construct the vertical constraint graph G_y*;*

if graph G_y is symmetric y-feasible

// if G_y has no cycles

compute the y-coordinates of the blocks;

// The O-tree is symmetric feasible

return the new placement configuration to the simulated annealing;

return to simulated annealing: O-tree symmetric infeasible

III.4 Experimental Results

The placement algorithm described in this chapter has been implemented in C on a SUN ULTRA-60 workstation.

Our benchmark tests are component blocks of a 2.4 GHz digital spread spectrum transceiver, entered recently in production at Conexant Systems Inc. This fully integrated transceiver device offers a complete RF system solution for 2.4 GHz cordless telephone applications, wireless LANs, and wireless modems.

Figure III.6 shows a placement of a small circuit with 14 cells. This placement is obtained in 0.17 minutes. There is one symmetry pair and one self-symmetric cell in this test cases. The placement is packed very tightly.

Figure III.7 shows the block placement of a circuit containing a symmetry group, consisting of 3 pairs of symmetric devices and 2 self-symmetric cells, which can be noticed near the bottom of the figure. The self-symmetric cells are cells presenting geometrical symmetry and sharing the same (horizontal) axis with the other symmetry pairs in the group. The 47 block example has been processed in 4.37 minutes. The layout area is only about 12% larger than the total area of all the blocks in the circuit, which proves the good rectangle packing capability of the placement tool.

Figure III.8 shows the layout (after placement) of another circuit characterized by a larger number of symmetric constraints. Our placement tool has processed this example (having 11 symmetry pairs and 1 self-symmetric cell) in only 5.21 minutes, which is an excellent result. The speed exhibited by the tool can be justified as follows: the upper-bound of the number of configurations in the O-tree representation [4] is smaller than the corresponding upper-bounds in other topological representations

(sequence-pair, BSG).

Table III.1 displays the placement results obtained for several component blocks of the 2.4 GHz transceiver. The smallest example having 14 cells runs in 10 seconds, while the largest example - with 110 cells - has been successfully placed in about 25 minutes. The placement results are relevant, as analog circuits seldom contain more than 100 cells per hierarchical level.

Our experiments suggest that the speed performance of our placement tool based on the O-tree representation is better relative to similar tools where placement configurations are represented employing absolute coordinates [10], or tools based on the sequence-pair topological representation [1].

Design	# of cells	Symmetry constraints	Area(μm^2)	Time (min)
vd2	14	1 symmetry pair, 1 self-symmetric cell	1365.04	0.17
dffrsdch	37	4 symmetry pairs	6286.93	1.89
tx_current	47	3 symmetry pairs, 1 self-symmetric cell	46309.50	4.73
lpf2_b25b	52	11 symmetry pairs, 1 self-symmetric cell	36245.88	5.21
dcservo_cmfb	66	3 symmetry pairs, 3 self-symmetric cells	60466.26	11.43
biasynth2p4g	67	8 symmetry pairs	4967.31	13.54
lnamixbias2p4G	110	4 symmetry pairs	49027.89	24.36

Table III.1: Experimental results for symmetry constraints packing

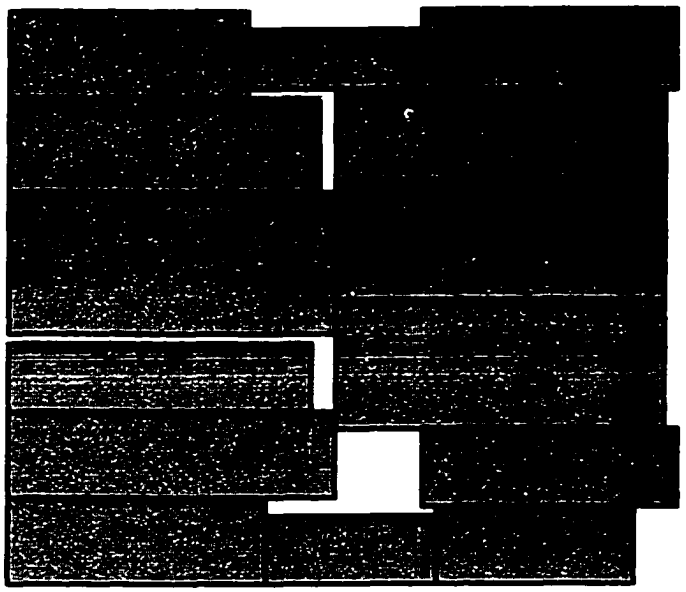


Figure III.6: Circuit vd2

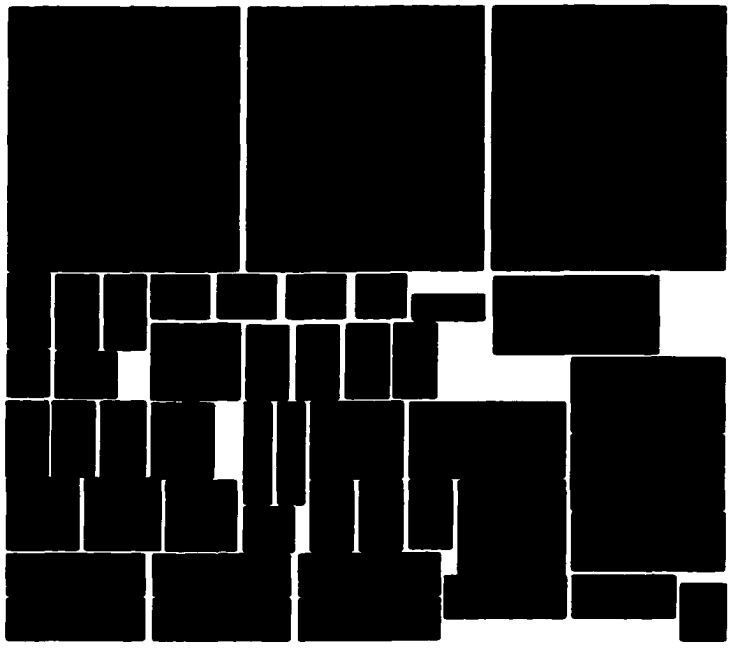


Figure III.7: Design tx_current with 47 cells

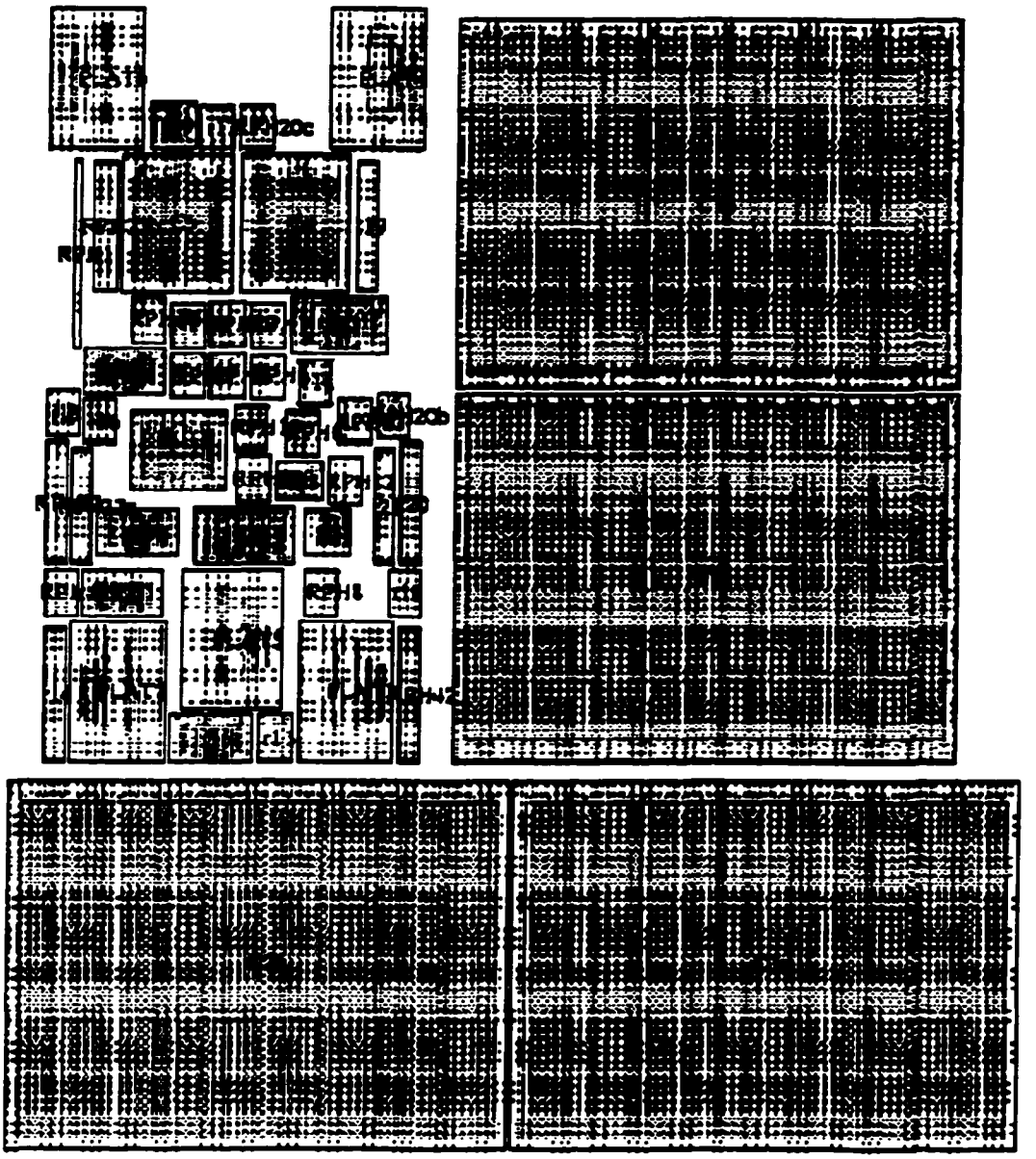


Figure III.8: Design *l*pf2_b25b with 52 cells

III.5 Summary

This chapter has addressed the problem of taking into account symmetry constraints when non-slicing floorplans are represented with O-tree structures. The necessity of handling symmetry emerges when, for instance, the O-tree topological representation is applied to device-level placement for analog layout. The good performance of our ordered tree-based placement tool when dealing with several analog designs taken from industry substantiates the effectiveness of our novel technique.

Portions of this chapter is a reprint of the material as it appears in "Block Placement with Symmetry Constraints Based on the O-tree Non-Slicing Representation", Yingxin Pang; Florin Balasa; Koen Lampaert; Chung-Kuan Cheng, 37th. Proceedings of ACM/IEEE Design Automation Conference, 2000. The dissertation author was the primary investigator and single author of this paper.

Chapter IV: Rectilinear Block Packing

IV.1 Introduction

With the recent advent of deep submicron technology and new packaging schemes, integrated circuit components are not limited to rectangular blocks. Traditional placement algorithms designed primarily for placing rectangles are no longer effective. Therefore new approaches which can efficiently handle rectilinear shaped blocks are essential in VLSI design.

Several methods to solve rectilinear block packing problem using slicing structure [27], and nonslicing structures such as sequence pair [6,12,33], and BSG [28] have been proposed. Each rectilinear module is partitioned into a set of rectangular subblocks which are individually handled as unit blocks. The relationship of the subblocks of a rectilinear module are represented in a encoding or in additional constraints. Therefore only the encodings that can lead to a feasible solution are considered. Generally the unit blocks are compacted in horizontal and vertical dimensions, and post processes are needed to restore the original shape of rectilinear blocks. The limitation of such a technique [12, 33, 28] is that when the number of blocks is large, the post processing is very hard to handle, and consumes a lot of time. Fujiyoshi *et al* [6] avoid this post processing problem by representing the necessary and sufficient condition of feasible sequence-pair directly in the horizontal and vertical constraint graphs. Then the packing determined by the horizontal and vertical constraint graphs satisfies both the constraints of the sequence pair and the rectilinear blocks and no post process is needed. However, this representation needs much time in constructing the relative position constraints, especially when the shape of the rectilinear block is complicated.

Based on the O-tree representation, this chapter proposes a new method to represent rectilinear shaped modules. Our representation has the following major properties:

(1) completeness: for each placement there is a representation corresponding to it
(2) Efficiency: the complexity of transformation from a representation to a feasible floorplan is low.

We first explore the L-shaped blocks and derive the L-admissible condition for an O-tree. This condition is sufficient and also necessary for a compact placement. Thus the non-L-admissible O-tree is redundant in the sense that if it can lead to a feasible solution after post alignment process, there exists another L-admissible O-tree which leads to the exact placement. Another advantage of our algorithm is that the process of restoring the original shape of the L-blocks and the placement are synchronized, so no post process is needed. Our method can be easily extended to handle the general rectilinear blocks by decomposing them into a set of sub L-shaped blocks. Instead of using the search method of simulated annealing, we use a heuristic optimization algorithm to search the optimal solution. The good performance of our placement algorithms when dealing with several examples proves the effectiveness of our approach.

The rest of the chapter is organized as follows: Section IV.2 discuss the algorithms for solving the L-shaped block packing problem. Section IV.3 presents a new optimization searching technique. Afterwards, Section IV.4 extends the algorithms to handle the general rectilinear block packing problem. Section IV.5 gives an overview of our experimental results, and finally, Section IV.6 presents a summary of this chapter.

IV.2 L-shaped Block Packing

In the rectilinear block packing, L-shaped blocks have the most important role, since many pre-destined library macro-cells are L-shaped block and rectilinear blocks can be decomposed into L-shaped blocks. Finding an efficient method to handle the L-shape block is extremely important in the rectilinear block packing problem.

An L-shaped block is a block whose shape is L. There are eight possible configurations for each L-shaped block as shown in Figure IV.1. Since a L-shape block can always be partitioned into two sub-rectangle blocks along the horizontal direction, the two sub-blocks will be aligned on as the left edge or the right edges. We call the L-shaped block whose sub-blocks are left aligned as *type I* L-shaped block, whose sub-blocks are right aligned as *type II* L-shaped block.

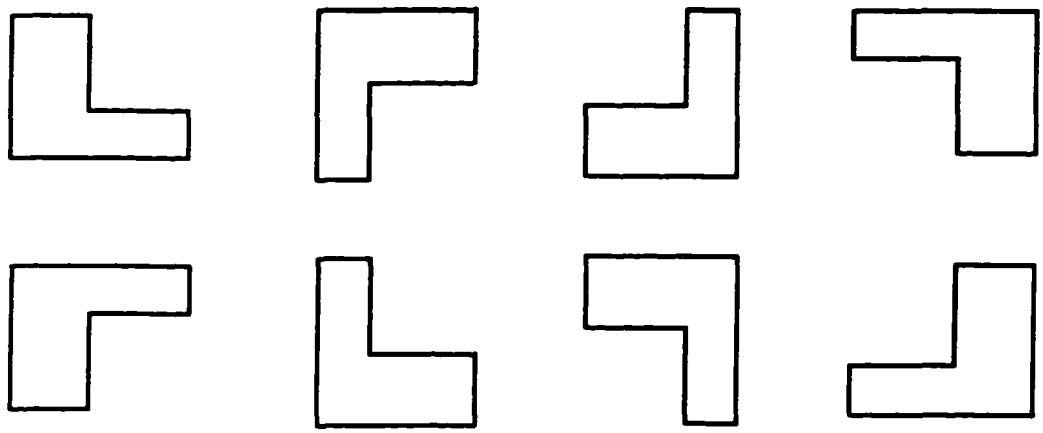


Figure IV.1: Eight configurations of an L-shaped block

IV.2.1 L-admissible O-tree

Assuming an L-shaped block is horizontally partitioned into two rectangle blocks A and B, then A and B are individually represented in a O-tree as unit blocks. Given a subset of the placeable cells constituting L-shaped blocks, not all the O-trees are feasible any more. The shape constraints may not be consistent with the O-tree encoding constraints. The relationship of an O-tree and a feasible placement is what we are interested in.

A type I L-shaped block can be easily handled by arranging its sub-blocks as adjacent siblings in the O-tree. The arrangement is not easy for a type II L-shaped block since the right alignment is needed and its two sub-blocks can be in two branches of the O-tree, each one can be the dominant block (which have the large coordinate of

the right edge).

Let $OT(T, \pi)$ be an O-tree containing rectangle blocks and type I and type II L-shaped blocks, the O-tree is defined to be *L-admissible* if:

(1) if $L(A,B)$ is a type I L-shaped block, A and B are adjacent siblings

(2) if $L(A,B)$ is a type II L-shaped block, x_A and x_B are the x-coordinates of blocks obtained by the original O-tree rule

if $x_B+w_B > x_A+w_A$, A has no children in the tree and there are no blocks between A and B in π whose horizontal spanning interval overlaps with $(max(x_B+w_B-w_A, x_B), x_B+w_B)$

if $x_A+w_A > x_B+w_B$, B has no children and there are no blocks between A and B in π has horizontal interval spanning overlapping with $(max(x_A+w_A-w_B, x_A), x_A+w_A)$

if $x_A+w_A = x_B+w_B$, there are no blocks between A and B in π has horizontal interval spanning overlapping with $(max(x_A, x_B), x_A+w_A)$

The O-tree shown in Figure IV.2 is L-admissible O-tree. On the other hand, the endcoding shown in Figure IV.3 is not L-admissible since b_{41} and b_{42} which are two sub-blocks of a L-shaped block do not satisfy both condition (1) and (2).

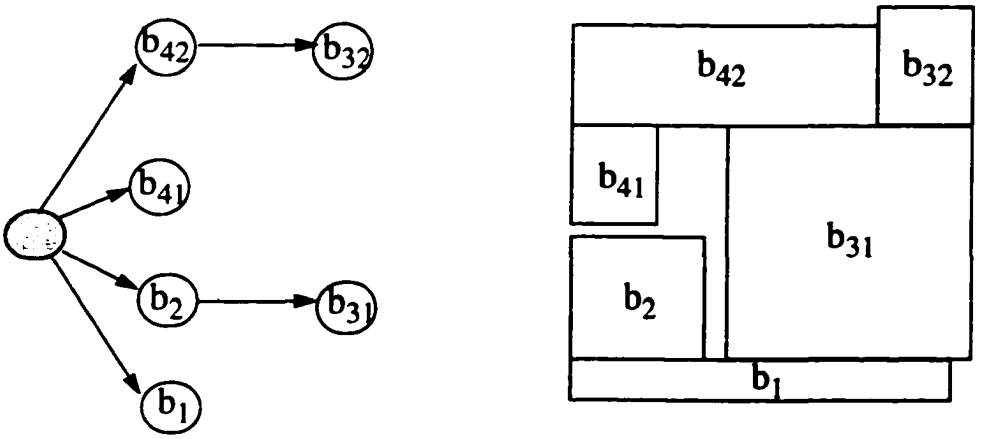


Figure IV.2: An L-admissible O-tree and its placement

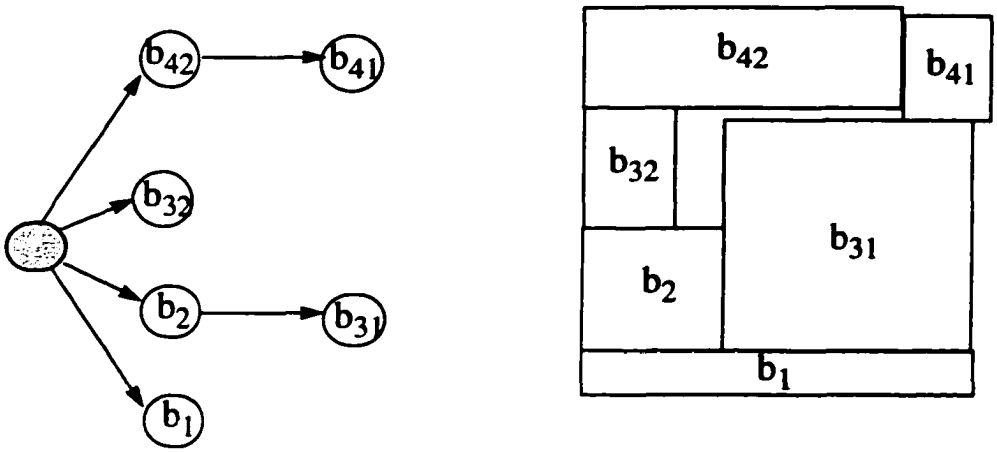


Figure IV.3: A non-L-admissible O-tree

If an O-tree is L-admissible, the following algorithm can transform it to a feasible placement.

Algorithm 1--L-shaped blocks packing

(1) *by the O-tree packing rule:*

$x_i = x_{parent\ of\ i} + w_{parent\ of\ i}$

(2) for every type II L-shaped block L(A,B)

if $x_B + w_B > x_A + w_A$, $x_A = x_B + w_B - w_A$

else $x_B = x_A + w_A - w_B$

(3) place the blocks by the depth first search order

let $\psi(i)$ *be the set of blocks index k lower than i in permutation* π,

whose spanning intervals $(x_k, x_k + w_k)$ *overlap* $(x_i, x_i + w_i)$. *Then*

$y_i = max_{k \in \psi(i)} y_k + h_k$ *if* $\psi(i)$ *is non-empty*

$= 0$ *otherwise*

(4) for every L-shaped block L(A,B), if A has lower index order in π

$y_A = y_B - h_A$

Theorem 1: *Given a set of placeable cells containing L-shaped block and a L-admissible O-tree, then one can build in linear time an optimal placement configuration satisfying the positioning and L-shape constraints.*

Proof: Given an L-admissible O-tree, a placement can be obtained by applying Algorithm 1. We will show that this construction has the properties stated in Theorem 1.

First, the x coordinates of the cells are computed by Step (1) and (2). Step 1 determines the x coordinates of the cells in $O(n)$ time such that the horizontal positioning constraints resulting from the O-tree are satisfied since it applies the original O-tree packing rule. In the absence of L-shaped block, the computation of the x coordinates is over; otherwise, the x coordinates must be trimmed in order to satisfy the L-shaped condition. This is done in Step (2). The type I L-shaped blocks are aligned automatically since the sub-blocks of a L-shaped block are adjacent sibling in the O-tree. Step (2) is to align the type II L-shaped blocks by pushing the sub-block with smaller right boundary rightward to align with the other sub-block. Because the O-tree is L-admissible, the sub-blocks with smaller right boundaries have no children, L-admissibility ensures the correct x-positioning constraints after step (2). Therefore this part of the algorithm satisfies the horizontal positioning constraints specific to the given O-tree and also satisfies the horizontal L-shaped constraints.

The y-coordinates are computed in step (3) and (4). The y-coordinates obtained by (3) satisfy the O-tree vertical constraints and Step (4) operates the abutment for L-shaped blocks. Assume $L(A,B)$ is a L-shape block and A has lower index order than B in the O-tree permutation π. If L is type I L-shape block, A and B are adjacent siblings. The blocks between A and B in the O-tree permutation π are the descendents of block A which are placed right to A. Therefore there are no blocks placed between A and B in the vertical direction. The lifting operation for A in step (4) restores the L-shape and does not affect any other blocks. If L is type II L-shape block, by the L-admissible condition, we know that no block is placed between A and B either. Therefore the lifting

in step (4) does not overlap with any other blocks and the L-shapes are restored. Overall, the placement is a feasible solution.

Steps (1) to (4) all require at most $O(n)$ operation, where n is the number of blocks. In conclusion, the algorithm described above runs in linear time.

Figure IV.2 shows a placement of an L-admissible O-tree by applying Algorithm 1.

Theorem 2: *The x-coordinates of the rectangular blocks in the feasible solution of a L-admissible O-tree are not altered by restoring the shape of the L-shaped blocks.*

Proof: In Algorithm 1, the x-coordinates of the blocks are determined by steps (1) and (2). In step (1) the x-coordinate of the each block is obtained by the O-tree packing rule without considering any L-shape constraints. Step (2) modify only those blocks which associated with L-shape. Therefore the x-coordinate of the rectangle blocks are not altered by our algorithm.

Remark: If we use special stack structure, the verification of the L-admissible condition can be obtained in linear time.

IV.2.2 Completeness and Compactness of the L-admissibility

A placement is compact if and only if there is no block that can be shifted to the left or shifted down from its original position with other components fixed. The optimal placement is compact. In order to find a placement with minimum area, it is sufficient to explore the subspace of L-admissible O-trees, rather than the entire O-tree space, which is significantly larger. The following theorem gives the sufficiency of the L-admissible O-trees.

Theorem 3: *Any compact placement configuration containing the L-shaped*

block can be encoded with an L-admissible O-tree.

Proof: Based on the placement, the horizontal constraint graph can be generated. The weight for an edge (B_i, B_j) is the separation distance between two node,

$$weight\ (B_i, B_j) = x_j - x_i - w_i .$$

The weight is equal to *0* when there is no empty space between the two blocks in horizontal direction. Since there are no edges crossing other edges, the horizontal constraint graph is planar. The shortest path tree can be constructed by running the shortest path algorithm of the graph. The shortest path tree is used as our initial O-tree. Each L-shaped block is partitioned into two sub-blocks along the horizontal direction. We can represent it as *L(A,B)*, where *A* is the bottom sub-block and *B* is the top one. If *A* and *B* are left aligned in the placement, *L* is a type I L-shaped block. We delete the original incoming edge of *A* in the O-tree and add an incoming edge to it from the parent of *B*. Therefore the two sub-blocks of a type I L-shape block become adjacent siblings in the O-tree.

If *A* and *B* are right aligned in the placement, *L* is a type II L-shaped block. Because the placement is compact, either A or B or both are compact to the left.

(1) If *B* is compact to the left in the compact placement, remove all children of *A*, and add them as the children of *B*, Those new children must keep original order and have higher index order than original children of *B* in the O-tree permutation.

(2) If A is compact to the left in the compact placement, remove all children of *B*, and add them as the children of *A*, Those new children must keep original order and have higher index order than original children of *A* in the O-tree permutation.

The trimmed O-tree satisfies the condition of L-admissible O-tree, and it can produce the same placement by our Algorithm 1.

A non-L-admissible O-tree, may lead to a feasible placement by pushing blocks rightward and upward. For an example, given non-L-admissible O-tree shown in Figure IV.4(a), a placement without considering any L-shaped constraints is shown in

Figure IV.4(b). In order to restore type II L-shape block $B(b_1,b_2)$, we operate a post alignment process by pushing block b_2 and c rightward. The resulting placement shown in Figure IV.4(c) is feasible. But this placement can be obtained by another L-admissible O-tree shown in Figure IV.4(d). In this sense, the non-L-admissible O-trees are redundant. Therefore, excluding non-L-admissible O-trees does not reduce the feasible solution space.

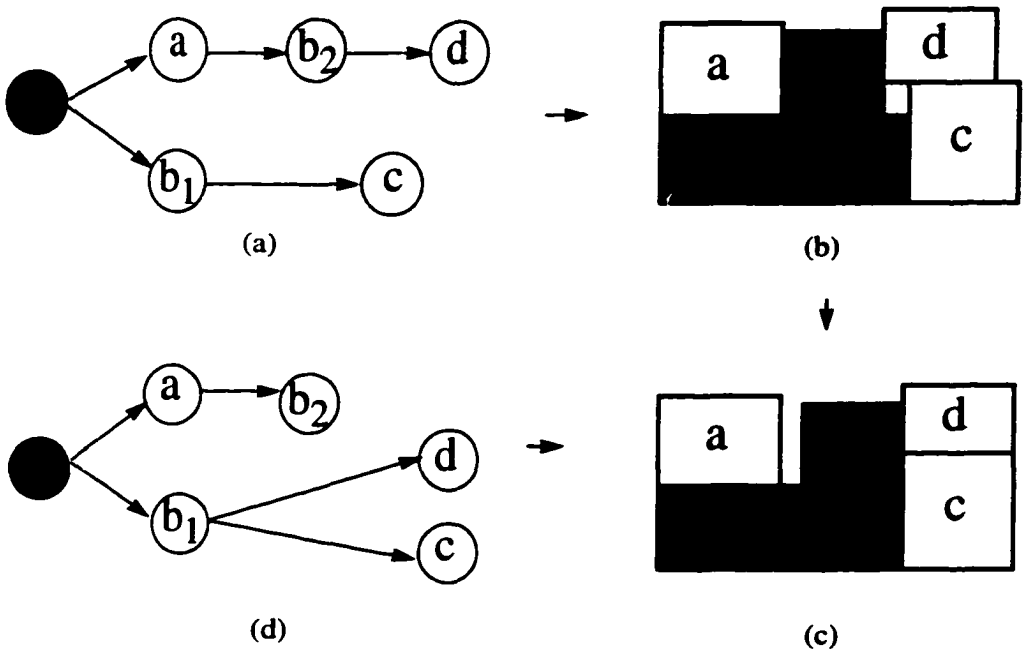


Figure IV.4: The redundancy of the non-L-admissible O-tree

IV.2.3 Stochastic Search

Among all O-trees, there are L-admissible O-trees including the optimal solution and non-L-admissible O-trees. The simulated annealing can be easily adapted to explore the solution space. We should consider how to treat the non-L-admissible O-trees when exploring the solution space by a stochastic algorithm. The simplest way would be to disregard it. Unfortunately, such a simple solution is not very effective, since a large percentage of time is wasted in those non-L-admissible O-trees which

would be rejected. An alternative way is to keep the smoothness of the search by adapting the infeasible one to a feasible one. For this purpose, we present a adaptation procedure which transforms an non-L-admissible O-tree to a L-admissible O-tree by only changing the positions of the sub-blocks associated with L-shaped blocks.

Algorithm 2 -- adaptation

Input: An infeasible O-tree(T[1:2n], π[1:n])
Output: A feasible O-tree
for each type I L-shaped block
if the two sub-blocks are not adjacent siblings
randomly delete one
insert it as the other one's adjacent sibling
end for
for i=1 to n
obtain the x-coordinates of all blocks by the original O-tree rule
if π[i] is a sub-block of type II L-shaped block and its
corresponding sub-block B has higher index order in π
for (k=i; k!=B; k++)
if (x[k], x[k]+w[k]) overlaps (x[i]+w[i]-w[B],x[i]+w[i])
delete B
insert it as the lower adjacent sibling of π[k]
end for
end for

It is well known that general stochastic search demands an enormous computation effort. Based on the enhanced perturbing algorithm in Chapter II, we present a heuristic search algorithm for the L-shaped block packing in next section.

IV.3 Heuristic Optimization Algorithm

IV.3.1 Four Direction O-trees

Given an O-tree, we place it according to the packing rule in chapter I section 3.1: the root corresponds to the left-bottom corner of the chip, the x-coordinate of a block is right to its parent and as left as possible, and the y-coordinate is as down as possible. Therefore the O-tree is placed along the left bottom boundary of the chip. We specify it as the Left-Bottom Horizontal O-tree(*LBHOT*). Based on the Left-Bottom Horizontal O-tree, we generate three other direction O-trees. They are the Bottom-Right Vertical O-tree(*BRVOT*) whose y-coordinates are determinated by the O-tree packing rule and is placed along the right and bottom boundary, the Right-Top Horizontal O-tree(*RTHOT*) whose x-coordinates are determined by the O-tree packing rule and is placed along the top and left boundary; and the Top-Left Vertical O-tree(*TLVOT*) whose y-coordinates are determined by the O-tree packing rule and is placed along the top and left boundary. Figure IV.5 shows the four different O-trees generated from the O-tree in Figure IV.1.

If B_1, B_2, ... , and B_n are the blocks to be placed on a chip, assume each B_i is a rectangle, having associated with it a width w_i and height h_i , and having (x_i , y_i) as coordinates of its left-bottom corner. The placements of the four direction O-trees are determined by the following rules:

The left-Bottom Horizontal O-tree: *LBHOT*

It is the general O-tree, where the rule of its placement is introduced in Chapter I.

The Bottom-Right Vertical O-tree: *BRVOT*

- the root may be viewed as the right bottom corner of the chip boundary,

  x_{root} = *Width of the chip*

  $y_{root} = 0$

  $w_{root} = 0$

  $h_{root} = 0$

- if B_i is the parent of B_j, then $y_j = y_i + h_i$.

- for each block B_i, let $\psi(i)$ be the set of blocks B_k which have been placed on the chip and whose spanning interval $(y_k, y_k + h_k)$ overlaps $(y_i, y_i + h_i)$. Then

$$x_i = \begin{cases} Max_{k \in \psi(i)}(x_k - w_i) & \text{if } \psi(i) \text{ is nonempty} \\ Width\ of\ the\ chip & \text{otherwise} \end{cases}$$

The Right-Top Horizontal O-tree: *RTHOT*

- the root may be viewed as the right top corner of the chip boundary,

$x_{root} = Width\ of\ the\ chip$

$y_{root} = Height\ of\ the\ chip$

$w_{root} = 0$

$h_{root} = 0$

- if B_i is the parent of B_j, then $x_j = x_i - w_i$.
- for each block B_i, let $\psi(i)$ be the set of blocks B_k which have been placed on the chip and whose spanning interval $(x_k, x_k + w_k)$ overlaps $(x_i, x_i + w_i)$. Then

$$y_i = \begin{cases} Max_{k \in \psi(i)}(y_k - h_i) & \text{if } \psi(i) \text{ is nonempty} \\ Height\ of\ the\ chip & \text{otherwise} \end{cases}$$

The Top-Left Vertical O-tree: *TLVOT*

- the root may be viewed as the left top corner of the chip boundary,

$x_{root} = 0$

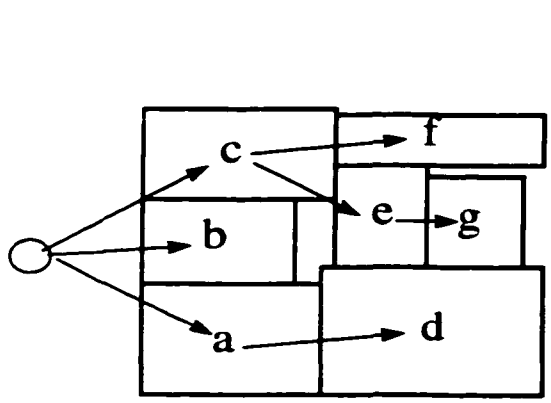
$y_{root} = Height\ of\ the\ chip$

$w_{root} = 0$

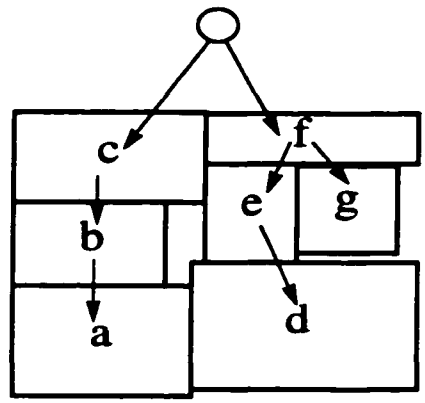
$h_{root} = 0$

- if B_i is the parent of B_j, then $y_j = y_i - h_i$.
- for each block B_i, let $\psi(i)$ be the set of blocks B_k which have been placed on the chip and whose spanning interval $(y_k, y_k + h_k)$ overlaps $(y_i, y_i + h_i)$. Then

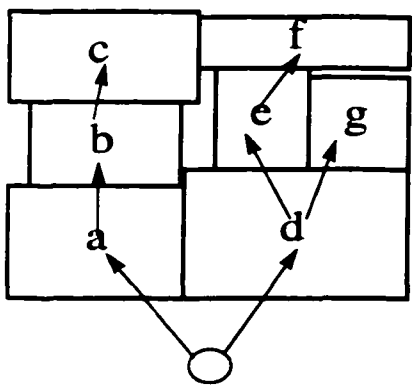
$$x_i = \begin{cases} Max_{k \in \psi(i)}(x_k + w_i) & \text{if } \psi(i) \text{ is nonempty} \\ 0 & \text{otherwise} \end{cases}$$



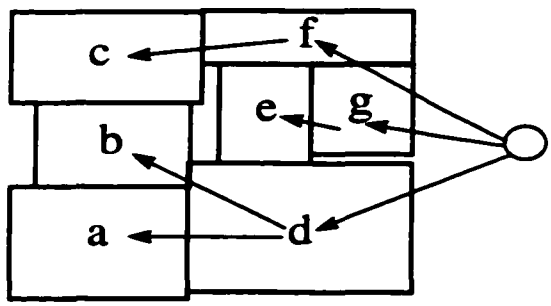
Left-Bottom Horizontal O-tree
(00110100011011, adbcegf)



Top-Left Vertical O-tree
(00011100011011, cbafedg)



Bottom-Right Vertical O-tree
(00100111000111, dgefabc)



Right-Top Horizontal O-tree
(00110011001011, fcgedba)

Figure IV.5: Four direction O-trees

Given an O-tree, its four direction O-trees can be constructed one by one in counterclockwise order and each one can be generated from the placement of the previous one.

Given a horizontal O-tree, its orthogonal O-tree can be obtained in the process of its placement. In the placement, the x-coordinates of the blocks are determined by the parent and children relationship. In order to reduce the run time for finding the y-coordinate of a block, a contour structure is used. For each newly added block B_i, we start from its parent in the contour, and keep on tracing the contour until we reach an x-coordinate is larger than x_i+w_i, where x_i is the x-coordinate of B_i, w_i is the width of B_i. Since the block is placed as far down as possible, there is one block on the traced contour that has a maximum top boundary. Then we add a new edge in the orthogonal O-tree which links these two blocks. In this way, we add the new block into the orthogonal O-tree. After the placement, all blocks are added in the orthogonal O-tree. The operation for finding orthogonal O-tree does not increase much complexity to the placement, therefore, the whole process is still in linear time. Figure IV.6 shows how to determine the y-coordinate of a new added block, and how to find the new orthogonal O-tree edge.

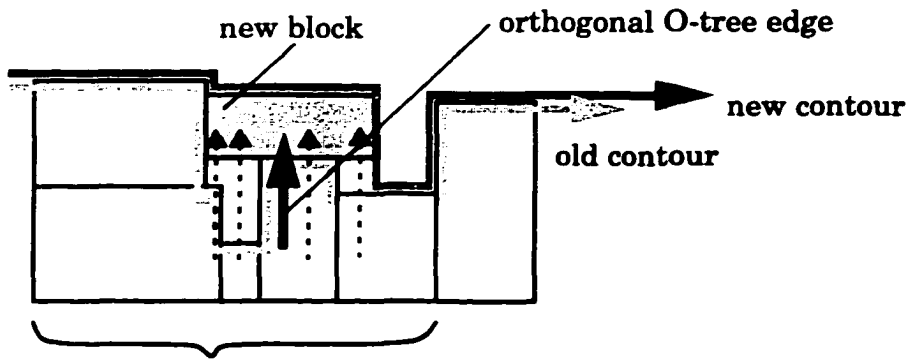


Figure IV.6: A new block is added to the placement

The orthogonal O-tree of a vertical O-tree can be constructed similarly. The orthogonal O-tree of a horizontal O-tree is a vertical O-tree, and the orthogonal O-tree of a vertical O-tree is a horizontal O-tree. If we produce the four direction O-trees in the order of *LBHOT, BRVOT, RTHOT, TLVOT*, then the *LBHOT* is the same as the given O-tree, the next direction O-tree is obtained by reversing all the sibling order of the orthogonal O-tree of the previous O-tree.

The horizontal partition of an L-shape is natural for Horizontal O-trees, whose L-shaped block is aligned on left or right edge. The vertical partition is natural for vertical O-trees whose L-shaped block is aligned on the bottom or top edges.

IV.3.2 Heuristic Optimization Algorithm

The intention of the construction of these O-trees is to thoroughly explore the local search. The different partition and the different alignment of L-shaped blocks correspond to different O-trees. A local adjustment from one O-tree to another may save the admissibility or improve the placement significantly.

The following is the basic outline of our heuristic algorithm:

Algorithm 3

Input: (i) A set of S, (ii) L={L-shape block in S}, |L|=l

(iii) R={Rectangle block in S}, |R|=r

Output: A floorplan of blocks in S

slicing each L-shape block into two sub-blocks,

each sub-block is treated as a unit rectangle block

randomly construct an 2l+r nodes O-tree OT(T, π)

while (j <= 2l+r)

A=π[j]

LBHOT=delete A from OT

if A is a subblock of an L-shaped block

LBHOT=delete the other sub-block B from LHOT

construct LBHOT's corresponding different direction O-trees:
BRVOT , RTHOT and TLVOT
insert A or (A, B) back to LBHOT, BRVOT, RTHOT, TLVOT
at the best possible inserting positions,
BestOT=min(cost(LBHOT), cost(BRVOT), cost(RTHOT), cost(TLVOT))
if cost(BestOT) < cost(OT)
OT=BestOT
j++
end while
output the L-admissible O-tree with minimum area

Remark In order to avoid trapped in a local optimal O-tree, the cost function that is used in the heuristic algorithm can be the weighted function of area and penalty of the non-L-admissibility. The weight of the penalty of the non-L-admissibility is increased exponentially so that we have only L-admissible configuration at later stages.

As in Chapter II, when we perturb an O-tree, we delete a node and insert it back to the O-tree only in the external insertion positions. Now we use the four direction O-trees; an inside insertion position of one O-tree may corresponding to an external insertion position of another O-tree. Therefore almost all possible insertion positions are tested in these four O-trees. The best insertion position for each O-tree can be obtained in linear time by following the depth first search sequence, reducing the checking of each position to constant time by amortizing the checking of other positions in that sequence. It is desirable that the insertion does not destroy the L-admissibility of the O-tree; therefore we only consider those positions which keep the L-admissibility. This can be done by checking the contour of the ceiling blocks and the floor blocks and that the insertion position can not be between two sub-blocks of a L-shape block. For each insertion position, we test four configurations of L-shape block

shown in the Figure IV.7 and IV.8. All eight configurations for a L-shaped block can all be tested in the four O-trees.

Theorem 4: *The insertion does not change L-admissibility of the original O-tree if we do not insert the block between two sub-blocks of a L-shaped block.*

Proof: Because we insert nodes as the external nodes of the O-tree, x-coordinates of the nodes in the original O-tree are not altered. The alignment properties are maintained. Furthermore, we do not insert in the positions between two sub-blocks of a L-shaped block. therefore the abutment property is not destroyed by this insertion. The new O-tree is admissible if the original O-tree is L-admissible.

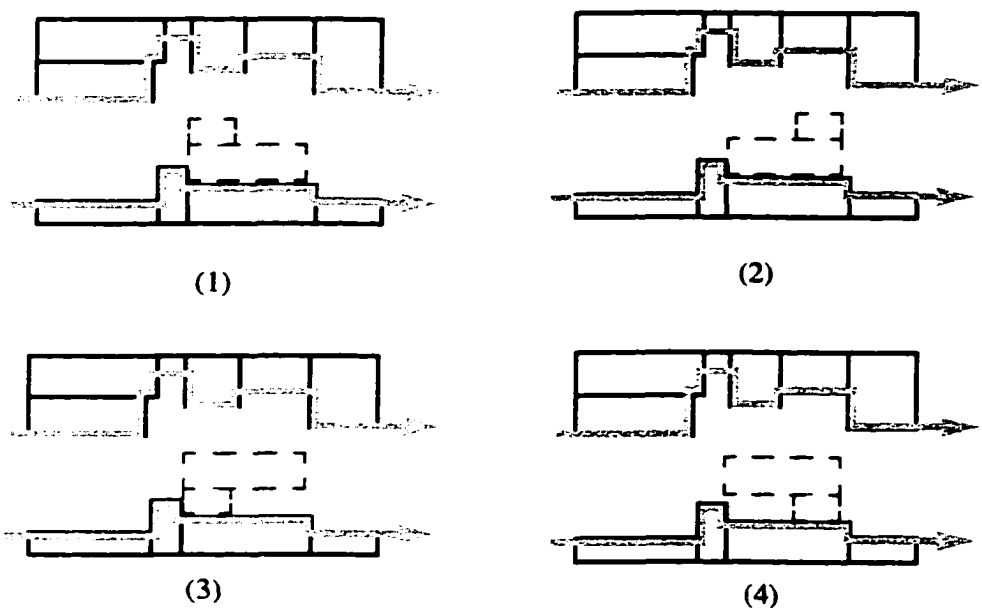


Figure IV.7: Four inserting configurations of an L-shape block in a horizontal O-tree

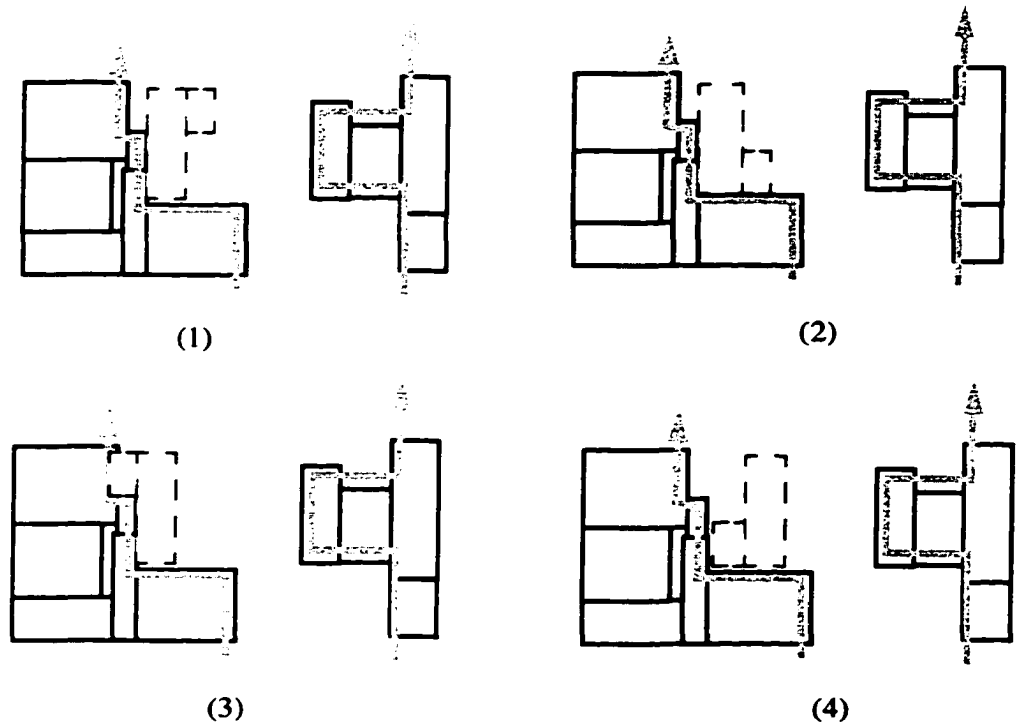


Figure IV.8: Four inserting configurations of an L-shape block in a vertical O-tree

IV.4 Rectilinear Block Packing

Let A denote an arbitrarily shaped rectilinear macro block. A can be partitioned into a set of rectangular sub-blocks by slicing A from the left to right along every vertical boundary of A. The partition is referred to as a horizontal partition. Similarly A can be vertically partitioned. Each sub-block is individually a rectangle block.

Assume the rectilinear block A is horizontally partitioned into a set of rectangular sub-blocks $(B_1, B_2, \ldots, B_k)$. Blocks B_i and B_j are called adjacent sub-blocks if they share a common slicing line. For any two adjacent sub-blocks B_i and B_j, if they form an L-shaped block, we say the rectilinear block is L-shape divisible. Otherwise we create a rectangle block B_{ij} which has very small height, and insert it in the middle of B_i and B_j, and make (B_i, B_{ij}) and (B_{ij}, B_j) form an L-shape. The process is called L-parti-

tioning and the L-shaped blocks are called sub-L-shaped blocks. Figure IV.10 shows an L-shape divisible rectilinear block, adjacent sub-blocks (B_1, B_2) and (B_2, B_3) form L-shaped blocks. Figure IV.11 shows a non-L-shape divisible rectilinear block and its L-partitioning.

Figure IV.9: L-shape divisible rectilinear block

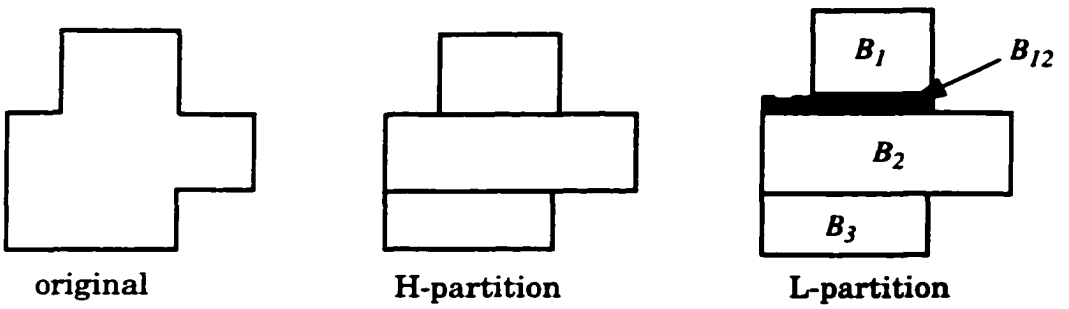


Figure IV.10: L-partitioning of a non-L-shape divisible rectilinear block

Since a rectilinear block can be L-partitioned into a set of sub-L-shaped blocks, the properties of L-shaped blocks can be extended to general rectilinear blocks.

If all rectilinear blocks are horizontally L-partitioned, we introduce the alignment constraints in the original O-tree constraint graph:

for any sub-L-shaped block $L(B_i, B_j)$ add edge (B_i, B_j) and (B_j, B_i)

if $L(B_i, B_j)$ is type I L-shaped, $w(B_i, B_j)=0$, $w(B_j, B_i)=0$

else $w(B_i, B_j)=w_i-w_j$, $w(B_j, B_i)=w_j-w_i$

An O-tree is admissible if it satisfies the following two conditions:

(1) there is no positive cycle in the extended constraint graph, and the x-coordinates of the blocks are the longest path length from the root to the nodes in the graph.

(2) for any type II sub-L-shaped block, there are no blocks between B_i and B_j in π has horizontal interval spanning overlapping with $(max(x_{Bi}, x_{Bj}), x_{Bi}+w_{Bj})$

Assume (B_{i-1}, B_i) and (B_i, B_{i+1}) are two adjacent sub-L-shaped-blocks of a rectilinear block. If there is no positive cycle in the extended constraint graph, the alignment property can be restored for both of them. Thus, the alignment property is restored for the rectilinear block (B_{i-1}, B_i, B_{i+1}). By the transitivity property, the alignment properties are restored for all rectilinear blocks.

A rectilinear block is called convex if any two points in the block have a shortest Manhattan path inside the block, as shown in Figure IV.11 (a). Otherwise the block is called concave, as shown in Figure IV. 12(a).

As in L-shaped block packing, the abutment operation for a rectilinear block does not alter the positions of the blocks which are not the sub-blocks of this rectilinear block. Therefore, only the blocks with smaller width are moved to abut the blocks with larger width.

if (B_{i-1}, B_i) and (B_i, B_{i+1}) are two adjacent sub-L-shaped-blocks of a convex rectilinear block. Then the width of B_i can not be smaller than both the width of B_{i-1} and B_{i+1}, therefore, the block B_i does not need to abut two blocks. The abutment property is restored for the rectilinear block (B_{i-1}, B_i, B_{i+1}). The transitivity property is also true for abutment for all convex blocks.

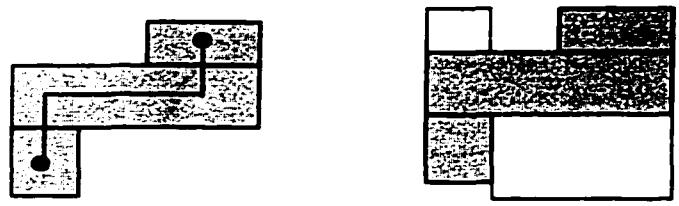


Figure IV.11: Convex rectilinear block packing

If $\{B_{i-1}, B_i\}$ and $\{B_i, B_{i+1}\}$ are two adjacent sub L-shaped blocks of a concave rectilinear block. It can be observed that the shape of some concave block cannot be restored by simply applying the abutment operation of our algorithm as shown in Figure IV.12. For those cases, we can either discard those packings or do the extension of some parts of the rectilinear block as shown in Figure IV.12 (c).

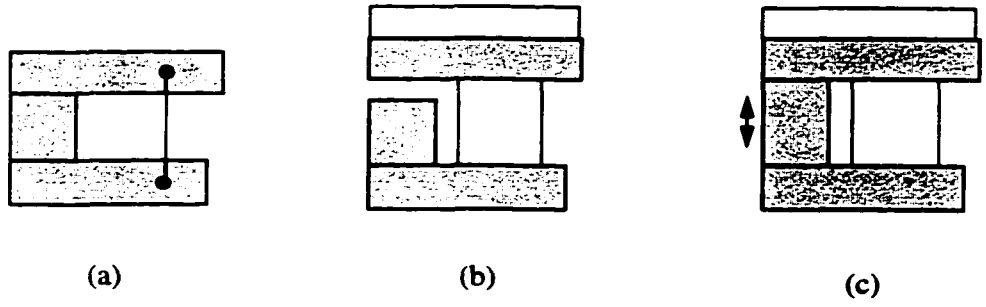


Figure IV.12: Concave rectilinear block packing

IV.5 Experimental Results

We have implemented the algorithm proposed in this paper using the C language and tested on Sun Ultra-Sparc 60 workstation.

Figure IV.13 shows a test case which has been processed by our algorithm in 12

seconds. It has eight blocks, where two of them are L-shaped blocks. The dead space is only 5.23%.

Figure IV.14 shows the placement of eighteen blocks, where six of them are L-shaped. The result is obtained in 130 seconds and the dead space is only 6.62%. In order to show the capability of handling the rectilinear block packing, we include some rectilinear blocks other than L-shape block in this test case. Now there are sixteen blocks, two of them are general rectilinear shape and five of them are L-shaped block . The placement is shown in Figure IV.15. The dead space is 9.21%.

Figure IV. 16 gives the packing of modified MCNC Benchmark ami49. It has 49 modules, seven of which are L-shaped blocks, it takes 140 seconds, and the dead space is about 9%. All the results show the effectiveness of our algorithm.

Figure IV.13: Test case 1 on rectilinear block packing

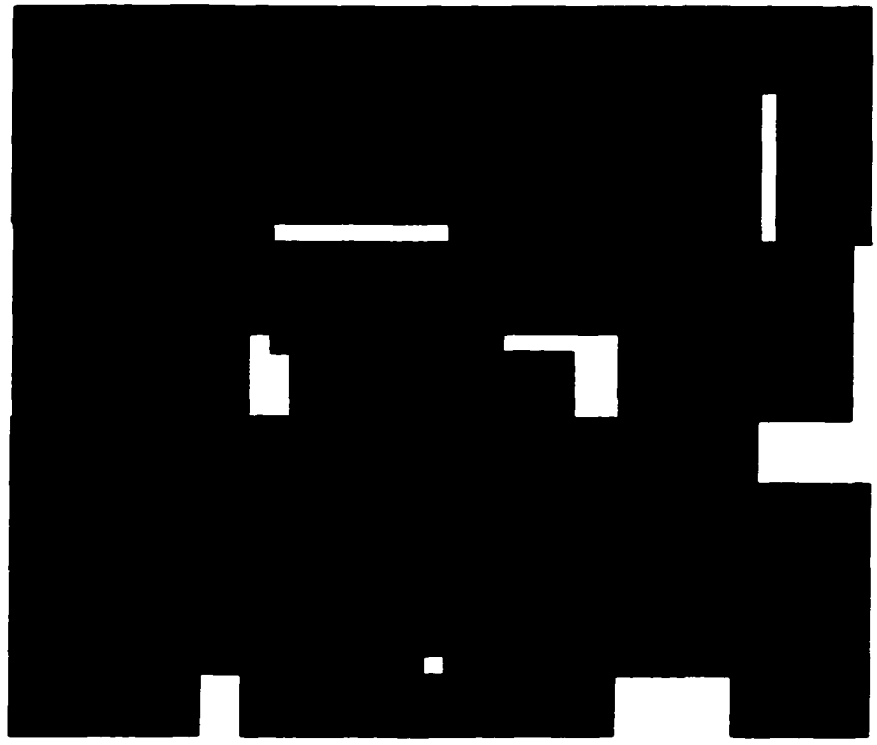


Figure IV.14: Test case 2 on rectilinear block packing

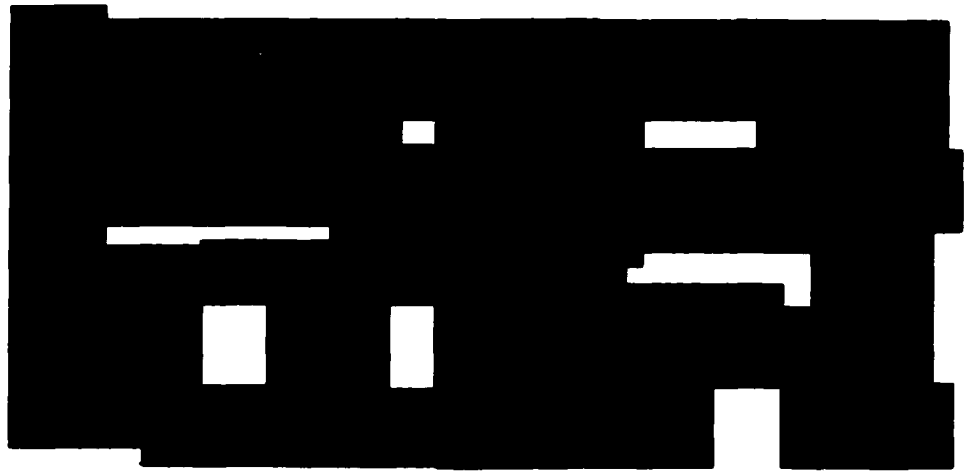


Figure IV.15: Test case 3 on rectilinear block packing

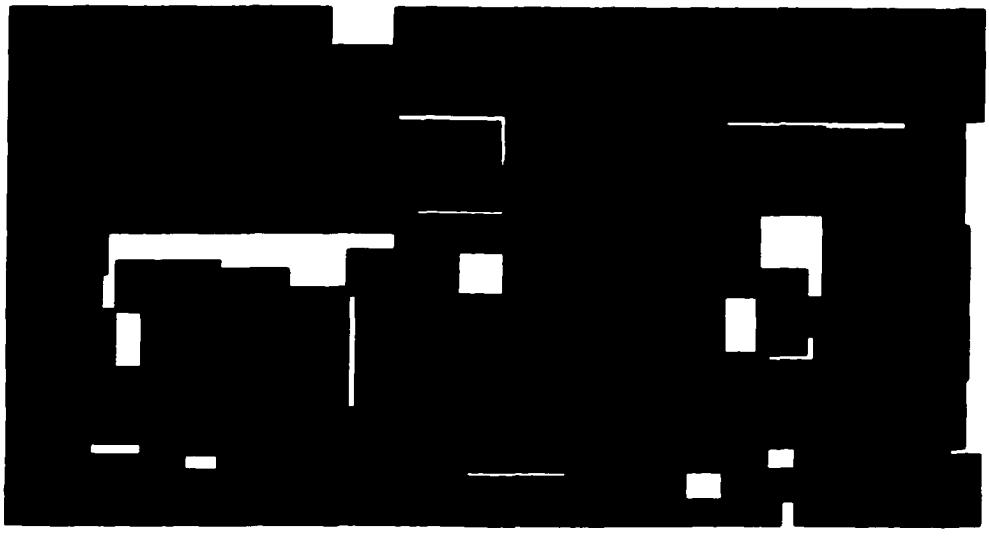


Figure IV.16: Test case 4 on rectilinear block packing

IV.6 Summary

In this chapter, we have addressed the rectilinear floorplanning problem in the context of the O-tree structure. We have defined the compact feasible solution space for L-shaped blocks and extended it to handle general rectilinear blocks. We have also presented a new optimization method, which combines the thorough local search and global randomization. It is efficient in area minimization for L-shaped block packing and general floorplanning. The experimental results are very promising.

Chapter V: Conclusions

As sub-micron technology dominates future VLSI systems, it is possible to realize a big system on a single chip. Designing such a huge VLSI layout is hard. Therefore it has inspired an immense literature of valuable research results. In this dissertation, we have described some floorplanning problems and presented solutions.

Floorplanning is a procedure to produce a floorplan (arrangement of objects in a configuration without overlaps) which is optimal with respect to a given objective. In this dissertation, we use a more recent topological representation called O-tree, which has several important advantages over other representations and show that it is adequate for solving general floorplanning problems.

We have presented an efficient algorithm to solve the core floorplanning problem. Given a placement configuration, a change to that configuration is made by moving a component heuristically to the best possible position. The best position is found by following a depth first search order sequence, reducing the time of checking each insertion position to constant time by amortizing the checking of other positions in that sequence. The proposed algorithm can also be extended to handle soft blocks. Our test results clearly show a significant reduction of timing while with good placement in terms of placement area and wire length.

We have addressed the problem of device-level placement for analog layout, focusing mainly on symmetry related aspects. Different from most of the existent analog placement approaches, our model adds the symmetry constraints in the constraint graph of O-tree directly. Therefore, the test of symmetric feasibility and the placement construction could be done simultaneously. Several analog examples substantiate the effectiveness of our placement tool, which is in use in an industrial environment.

We have proposed a new method to represent and pack rectilinear blocks. The L-shaped blocks are examined first. The properties of L-shaped blocks can be applied

to general rectilinear blocks by decomposing them into a set of L-shaped blocks. We give a simple representation for L-shaped blocks packing, it is efficient and complete. In order to explore the local search thoroughly, we produce four direction O-trees. A heuristic algorithm based on the four direction O-trees gives very good experiment results.

Over the years floorplanning has been an active research topic. A rapid decrease in device dimensions leads to much higher design complexity. The floorplaning algorithms must be able to scale well to match the rapid increase in design complexity. Since designing such a huge VLSI layout is hard and design reuse has been attracting much interest, the floorplanning algorithms should be able to handle L-shaped or T-shaped blocks.

In the future, the floorplanning algorithm must be flexible enough to match the change of the physical design and be able to handle uncertainties.

Bibliography

[1]F. Balasa, K. Lampaert, "Module Placement for Analog Layout Using the Sequence-Pair Representation", *Proc. 36th ACM/IEEE Design Automation Conf.*, pp. 274-279, June 1999.

[2]T. Chang Lee, "A Bounded 2D Contour Searching Algorithm for Floorplan Design with Arbitrarily Shaped Rectilinear and Soft Modules", *Proc. 30th ACM/IEEE Design Automation Conf.*, pp. 525-530, 1993.

[3]C.K. Cheng, E.K. Kuh, "Module Placement based on Resistive Network Optimization", *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. CAD-3, pp. 218-225, 1984.

[4]J. Cohn, D. Garrod, R. Rutenbar, L. Carley, "KOAN/ANAGRAM II: new tools for device-level analog lay-out", *IEEE J. of Solid-State Circuits*, Vol. SC-26, No. 3, pp. 330-342, March 1991.

[5]T. H. Cormen, C. E. Leiserson, R. L. Rivest, "Introduction to Algorithm", *The MIT press*, 1990.

[6]K. Fujiyoshi and H. Murata, "Arbitrary Convex and Concave Rectilinear Block Packing using Sequence-Pair", *Proc. ISPD*, pp. 103-110, 1999.

[7]P.-N. Guo, C.-K. Cheng, T. Yoshimura, "An O-tree representation of non-slicing floorplan and its applications", *Proc. 36th ACM/IEEE Design Automation Conf.* , pp. 268-273, 1999.

[8]T.C. Hu, E.S. Kuh, "VLSI Circuit Layout: Theory and Design", *IEEE Press*, 1985.

[9]E. S. Kuh, T. Ohtsuki, "Recent Advances in VLSI layout", *Proc. of the IEEE*, Vol. 78, No. 2, pp. 237-263, 1990.

[10]K. Lampaert, G. Gielen, W. Sansen, "A performance driven placement tool for analog integrated circuits", *IEEE J. of Solid-State Circuits*, Vol. SC-30, No. 7, pp. 773-780, July 1995.

[11]M. Z. Kang and W. Dai, "General Floorplanning with L-shaped, T-shaped and Soft blocks Based on Bounded Slicing Grid Structure", *Proc. ASP-DAC*, pp. 265-270, 1997.

[12]M. Z. Kang and W. Dai, "Arbitrary Rectilinear Block Packing Based on Sequence Pair", *Proc. ICCAD*, pp. 259-266, 1998.

[13]M. Z. Kang and W. Dai, " Topology Constrained Rectilinear Block Packing", *Proc. ISPD*, pp.179-186, 1998

[14]E. Malavasi, E. Charbon, G. Jusuf, R. Totaro, A. Sangiovanni-Vincentelli, "Virtual symmetry axes for the layout of analog IC's", *Proc. 2nd ICVC*, pp. 195-198, Seoul, Korea, Oct. 1991.

[15]E.Malavasi, E. Charbon, E. Felt, A. Sangiovanni-Vincentelli, "Automation of IC layout with analog constraints", IEEE Trans. on Compter-Aided Design of IC's and Systerms, Vol. 15, No. 12, pp. 1518-1524, Dec. 1996.

[16]H. Murata, K. Fujiyoshi, S. Nakatake, Y. Kajitani, "VLSI module placement based on rectangle-packing by the sequence-pair", *IEEE Trans. on Comp. Aided Design of* IC's *and Systems*, Vol. 15, No. 12, pp. 1518-1524, 1996.

[17]H. Murata, K. Fujiyoshi, M. Kaneko, "VLSI/PCB Placement With Obstacles Based on Sequence-Pair", *IEEE Trans. on CAD*, Vol. 17, No.1, pp. 60-18, 1998.

[18]S. Nakatake, K. Fujiyoshi, H. Murata, Y. Kajitani, "Module packing based on the BSG-structure and IC layout applications", *IEEE Trans. on Comp. Aided Design of IC's and Systems*, Vol. 17, No. 6, pp. 519-530, 1998.

[19]R. Otten, "Complexity and diversity in IC layout design", *Proc. IEEE Intn'l Symp. Circuits and Computers*, 1980.

[20]R. Otten, "Automatic Floorplan Design", *Proc. of 19th ACM/IEEE Design Automation Conf., pp.261-267,* 1982.

[21]Y. Pang, F.J. Liu, C.K. Cheng, "A new Method for Hierarchical Circuit Composition", ACM, *Intn'l Workshop on Timing issues in the Specification and Synthesis of Digital Systems*, pp. 71-76, 1999.

[22]Y. Pang, C.K. Cheng T. Yoshimura, "An Enhanced Perturbing Algorithm for Floorplan Design Using the O-tree Representation", *ACM Proc. ISPD*, pp. 168-173, 2000.

[23]J. Rijmenants, J.B Litsios, T.R. Schwarz, M. Degrauwe, "ILAC: an automated layout tool for analog CMOS Circuits," *IEEE J. of Solid-State Circuits*, Vol. SC-24, No.2, pp.417-425, April 1989.

[24]T. C. Wang, D.F. Wong, "An Optimal Algorithm for Floorplan Area Optimization", *27th Design Automation Conference*, pp. 180-186, 1990.

[25]T. C. Wang, D. F. Wang, "Efficient Shape Curve Construction in Floorplan Design", *IEEE Trans. on CAD,* pp. 356-360, 1991.

[26]D. F. Wong, C. L. Liu, "A new algorithm for floorplan design," *Proc. 23th ACM/ IEEE Design Automation Conf.* pp. 101-107, June 1986.

[27]D. F. Wong, and C. L. Liu, "Floorplan Design for Rectangular and L-shaped Modules," *Proc. ICCAD*, pp. 520-523, 1987

[28]K. Sakanushi, S. Nakatake, and Y. Kajitani, "The Multi-BSG: Stochastic Approach to an Optimum Packing of Convex-Rectilinear Blocks", *Proc. ICCAD*, pp. 267-274, 1998

[29]L. Stockmeyer, "Optimal Orientations of Cells in Slicing Floorplan Designs", *Information and Control*, Vol. 59, pp. 91-101, 1983.

[30]S. Wimer, I. Koren, I. Cederbaum, "Floorplans, planar graphs and layouts", *IEEE Trans. Circuits System*, Vol. 35, pp. 267-278, 1988

[31]S. Wimer, I. Koren, "Optimal Aspect Ratios of Building Blocks in VLSI", *IEEE Trans. on Computer-Aided Design*, Vol. 8. No. 2, 1989.

[32]J. Xu, P.-N. Guo, and C.-K. Cheng, "Cluster Refinement for Block Placement", *ACM/IEEE Proc. 34th Design Automation Conference*, pp. 101-107.

[33]J. Xu, P.-N. Guo, and C.-K. Cheng, "Rectilinear block placement using sequence-pair, " *IEEE Trans. on Comp.* Aided Design of IC's and Systems, Vol. 18, No. 4, pp. 484-493, 1999.

[34]F. Y. Young, D. F. Wong, "Slicing Floorplans with Range Constraints", *ACM ISPD*, pp. 97-102, 1999.